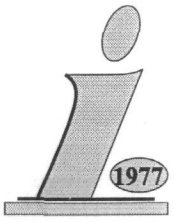
Volume 23 Number 1 April 1999 ISSN 0350-5596

Informatica

An International Journal of Computing and Informatics

Special Issue:
Parallel Computing on
Networks of Computers

Informatica 23 (1999) Number 1, pp. 1–152



The Slovene Society Informatika, Ljubljana, Slovenia

Informatica

An International Journal of Computing and Informatics

Basic info about Informatica and back issues may be FTP'ed from `ftp.arnes.si` in magazines/informatica ID: `anonymous` PASSWORD: <*your mail address*>
FTP archive may be also accessed with WWW (worldwide web) clients with
URL: `http://www2.ijs.si/~mezi/informatica.html`

Subscription Information Informatica (ISSN 0350-5596) is published four times a year in Spring, Summer, Autumn, and Winter (4 issues per year) by the Slovene Society Informatika, Vožarski pot 12, 1000 Ljubljana, Slovenia.
The subscription rate for 1999 (Volume 23) is
– DEM 100 (US$ 70) for institutions,
– DEM 50 (US$ 34) for individuals, and
– DEM 20 (US$ 14) for students
plus the mail charge DEM 10 (US$ 7).
Claims for missing issues will be honored free of charge within six months after the publication date of the issue.

LaTeX Tech. Support: Borut Žnidar, Kranj, Slovenia.
Lectorship: Fergus F. Smith, AMIDAS d.o.o., Cankarjevo nabrežje 11, Ljubljana, Slovenia.
Printed by Biro M, d.o.o., Žibertova 1, 1000 Ljubljana, Slovenia.

Orders for subscription may be placed by telephone or fax using any major credit card. Please call Mr. R. Murn, Jožef Stefan Institute: Tel (+386) 61 1773 900, Fax (+386) 61 219 385, or send checks or VISA card number or use the bank account number 900–27620–5159/4 Nova Ljubljanska Banka d.d. Slovenia (LB 50101-678-51841 for domestic subscribers only).

According to the opinion of the Ministry for Informing (number 23/216-92 of March 27, 1992), the scientific journal Informatica is a product of informative matter (point 13 of the tariff number 3), for which the tax of traffic amounts to 5%.

Informatica is published in cooperation with the following societies (and contact persons):
Robotics Society of Slovenia (Jadran Lenarčič)
Slovene Society for Pattern Recognition (Franjo Pernuš)
Slovenian Artificial Intelligence Society; Cognitive Science Society (Matjaž Gams)
Slovenian Society of Mathematicians, Physicists and Astronomers (Bojan Mohar)
Automatic Control Society of Slovenia (Borut Zupančič)
Slovenian Association of Technical and Natural Sciences (Janez Peklenik)

Informatica is surveyed by: AI and Robotic Abstracts, AI References, ACM Computing Surveys, Applied Science & Techn. Index, COMPENDEX*PLUS, Computer ASAP, Computer Literature Index, Cur. Cont. & Comp. & Math. Sear., Current Mathematical Publications, Engineering Index, INSPEC, Mathematical Reviews, MathSci, Sociological Abstracts, Uncover, Zentralblatt für Mathematik, Linguistics and Language Behaviour Abstracts, Cybernetica Newsletter

The issuing of the Informatica journal is financially supported by the Ministry for Science and Technology, Slovenska 50, 1000 Ljubljana, Slovenia.

Post tax payed at post 1102 Ljubljana. Slovenia taxe Percue.

Clustering–in Search for Scalable Commodity Supercomputing

The history of computing can be viewed as a constant search for computational power. As soon as a new, more powerful, computer is developed a larger problem to be solved appears on the horizon. This need for computational power, instead of leveling off, is growing day by day. During recent decades different high performance computer systems attempted to satisfy the power-hungry users. The most common systems were:

- Vector Computers (VC)
- Massively Parallel Processors (MPP)
- Symmetric Multiprocessors (SMP)
- Cache-Coherent Nonuniform Memory Access Computers (CC-NUMA)
- Distributed Systems

Although vector computers provided the breakthrough needed for the computational research to emerge as an independent science, they were only a partial answer as they could deliver top performance only for a few classes of problems. Many powerful scalable MPP systems have been built, but most of them have failed commercially due to their high cost and a low performance/price ratio. Symmetric multiprocessors are attractive, but they suffer from the scalability problem. Non-uniform memory access computers address some scalability and cost issues, but they suffer from a single point of failure as they use a single operating system kernel across all nodes (as in SMPs). Distributed systems are scalable but they do not offer ease of use or means for fast communication, which are essential requirements for efficient execution of parallel applications.

Recent years have witnessed a new direction of search for computational power – cluster computing (CC). (A cluster is a computer system that forms, in varying degrees, a single unified resource composed of several interconnected computers.) Although clustering or cluster computing has been around for more than 25 years it did not gain momentum until three technology trends converged in the 1980s: development of high performance microprocessors, emergence of high-speed networks and maturation of standard tools for high performance distributed computing. Another trend which is also worth mentioning in this context is the increased need for computing power in commercial applications coupled with the high cost and low accessibility of traditional supercomputers.

In recent years, the availability of high-speed networks and high performance microprocessors/workstations as commodity components make networks of workstations an appealing vehicle for cost-effective parallel computing. Clusters/networks of computers (workstations, PCs or SMPs) built using commodity hardware and software (such as Linux, PVM, or MPI) play a major role in redefining the concept of high performance computing. As a whole, clusters are becoming an alternative to MPPs and supercomputers in many areas of application.

This Special Issue is a result of an extremely large number of submissions that we received for the Special Issue of the *Parallel and Distributed Computing Practices (PDCP)* Journal [3]. Among more than five dozens of submissions, 24 papers have received very high recommendation from reviewers. We could not publish them all in *PDCP* but we were able to find a home for them here in *INFORMATICA*. A half of the selected papers will appear in PDCP and the remaining half appears in this issue. The focus of this Special Issue will be on both hardware and software aspects of cluster computing.

There are many ways of looking at cluster computing. Typically we consider them to be a number of machines physically connected via a wired network. This does not need to be the case in the future. We thus start from the paper by H. Zheng, et. al. *Mobile Cluster Computing and Timeliness Issues* which presents an overview of research issues involved in mobile cluster computing. In particular, they consider problems involved in cluster nodes migrating between cells of a wireless network. The next paper returns back to Earth and considers how the modern high-speed networks can be used to facilitate cluster computing. In *High-Performance Cluster Computing over Gigabit/Fast Ethernet*, J. Sang, et. al. consider cluster computing over Fast Ethernet and present practical results obtained using the NAS parallel benchmark. The last paper addressing the global issues is *The Remote Enqueue Operation on Networks of Workstations* by M. Katevenis, et. al. It contains an overview of communication mechanisms necessary to support cluster computing. In the paper a remote-enqueue atomic operation is introduced and compared with other possible alternatives.

The second group of papers is devoted to the load balancing and scheduling issues of clustering. In *Preserving Mutual Interests in High Performance Computing Clusters* O. Kremien et. al. address one of the drawbacks of the PVM environment (use of a simple round-robin process allocation policy) by adding to the environment a resource manager. This enables them to facilitate effectively the construction of clusters built of heterogeneous computers. Another approach to load balancing is presented by A. Bevilacqua in *A Dynamic Load Balancing Method on a Het-*

erogeneous Cluster of Workstations. His load balancing algorithm is based on the dynamic data assignment and its performance is studied for a 3D image reconstruction problem. In *Minimizing Communication Conflicts with Load-Skewing Task Assignment Techniques on Network of Workstations* W.-M. Lin and W. Xie study the load balancing problem for low speed networks. They present an algorithm which is particularly well suited for the bus-based communication. Finally, in *Scheduling of I/O in Multiprogrammed Parallel Systems*, P. Kwong and S. Majmudar address the effective management of parallel I/O for cluster computing. They develop a simulation model and compare the performance of various I/O scheduling strategies.

The next two papers are devoted to fault tolerance. D. Kebbal et. al. in *Fault Tolerance of Parallel Adaptive Applications in Heterogeneous Systems* discuss fault tolerance for heterogeneous adaptive systems. The proposed fault tolerance policy based on optimized coordinated checkpointing is shown to be an effective strategy allowing a recovery from failure by involving only a minimal part of the failed application. A fault tolerance approach based on utilization of idle cycles of computers in the cluster is proposed by T. Setz in *Fault Tolerant Execution of Compute-Intensive Distributed Applications in* LiPS. The proposed approach alleviates the need for application-wide synchronization used to generate sets of consistent checkpoints.

The last three papers are devoted to application development. E. Manolakos and D. Galatopoullos in JavaPorts*: An Environment to Facilitate Parallel Computing on a Heterogeneous Cluster of Workstations* shows the use of Java language for high performance computing. They demonstrate experimental results showing that a good performance can be achieved even on a relatively slow 10Mbs Ethernet based cluster of workstations. In *Structured Performability Analysis of Parallel Applications*, J. Dougherty presents a unified performance and dependability evaluation methodology for practical large-scale parallel applications. Experimental results comparing the performance obtained on a network of DEC Alpha Stations with the performance predicted by the theoretical model are presented. Finally, D. Helman and J. JaJa in *Sorting on Clusters of SMPs* discuss practical issues involved in developing an efficient sorting algorithm for a cluster of DEC SMP Alpha servers.

We would like to express our deep gratitude to Prof. M. Gams, Managing Editor of *INFORMATICA* who agreed to publish this Special Issue on a very short notice. This issue would not be possible without the help of referees (listed below) who worked very hard to review all the submitted papers. We would like to thank them all.

We hope you will find this special issue interesting.

Guest Editors

Rajkumar Buyya
Co-Chair
IEEE Task Force on Cluster Computing (TFCC)
School of Computer Science and Software Engineering
Monash University
Clayton Campus, Melbourne, Australia.
Email: *rajkumar@ieee.org*;
URL: http://www.dgs.monash.edu.au/~rajkumar/tfcc/

Marcin Paprzycki
Coordinator,
IEEE TFCC Tech. Area–Algorithms and Applications
Department Computer Science and Statistics
University of Southern Mississippi
Hattiesburg, MS 39406, USA
Email: *m.paprzycki@usm.edu*;
URL: http://orca.st.usm.edu/marcin/

References

[1] R. Buyya. *High Performance Cluster Computing: Systems and Architectures.* Volume 1, 1/e, Prentice Hall PTR, NJ, 1999.

[2] R. Buyya. *High Performance Cluster Computing: Programming and Applications.* Vol. 2, 1/e, Prentice Hall PTR, NJ, 1999.

[3] R. Buyya and C. Szyperski. *Special Issue on High Performance Computing on Clusters.* Parallel and Distributed Computing Practices (PDCP) Journal, Vol. 2 (2), June 1999.

[4] G. Pfister. *In Search of Clusters.* 2/e, Prentice Hall PTR, NJ, 1998.

[5] T. Sterling, J. Salmon, D. Becker, and D. Savarrese. *How to Build a Beowulf.* The MIT Press, 1999.

[6] B. Wilkinson and M. Allen. *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, NJ, 1999.

Reviewers

Alessandro Bevilacqua
Alfred Weaver
Amin Vahdat
Amitabh Dave
Amy Apon
Andrzej Goscinski

Benedict Gomes
Biersack Ernst
Boleslaw Szymanski
Boris Weissman
Cho-Li Wang
Chung-Ta King
Dan Hyde
David Bader
Davide Rossetti
Domenico Talia
Dorina Petriu
Dror Feitelson
El-ghazali Talbi
Erhan Saridogan
Gangan Agrawal
Gihwan Cho
Giuseppe Ciaccio
Harjinder Sandhu
Hye-Seon Meeng
Jamel Gafsi
Jay Fenwick
Jianyong Wang
John Dougherty
Kennith Birman
Lars Rzymianowicz
Lori Pollock
Luis Silva
Maciej Golebiewski
Mark Baker
Mark Clement
Marrianne Winslett
Mathew Chidester
Michele Colajanni
Orly Kremien
Paddy Nixon
Paul Roe
Putchong Uthayopas
Quinn Snell
Rainer Fraedrich
Rajeev Raje
Rajeev Thakur
Ricky Kwok
Robert Brunner
Robert Todd
Samuel Russ
Toni Cortes
Yong Cho
Yoshio Tanaka
Yu-Kwong Kwok

Mobile Cluster Computing and Timeliness Issues

Haihong Zheng, Rajkumar Buyya[1], and Sourav Bhattacharya
Dept. of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406, USA
Email: haihongz@asu.edu, rajkumar@ieee.org, sourav@asu.edu

Keywords: Cellular Network, Cluster Computing, Handover, Hypercube Topology, Mobile Cluster Computing, Mobile IP, Multicast, Rerouting, Timeliness, Triangle Mesh.

Edited by: Marcin Paprzycki
Received: December 15, 1998 **Revised:** February 25, 1999 **Accepted:** March 1, 1999

With the rapid advancement and extensive deployment of cluster computing and mobile communication, the integration of these two technologies has become feasible and lead to the emergence of a new paradigm called mobile cluster computing (MCC). Among the issues that need to be addressed before MCC can become a reality, the timeliness issue is an important one, especially when mobile nodes within a computing cluster migrate from one cell to another cell in a cellular wireless network. In this paper, we first define and analyze the potential application environment of mobile cluster computing. We also present a generic architecture of a mobile cluster computer and several potential research issues of mobile cluster computing. In the rest of this paper, we focus on the timeliness issue of routing and multicast when handover occurs, along with several solution approaches based on different system architectures.

1 Introduction

During the past decade many different computer systems supporting high performance computing have emerged. Among several common systems, clusters[2] have become increasingly popular to prototype, debug and run parallel applications [1]. Since individual workstations have become increasingly powerful, and the communication bandwidth between them is increasing and latency is decreasing, clusters can provide similar (or sometimes better) performance reliability as well as fault tolerance as the traditional mainframes or supercomputers.

In addition, we observe that, as a result of the rapid development of mobile wireless networking system, users can access information across distributed sites and exploit the capacities of the global network at any time without regard to the location or mobility of the end units. This suggests that in addition to stationary nodes, mobile nodes will enter the cluster computing arena and result is the emergence of a new paradigm called "Mobile Cluster Computing" (MCC).

Another boost to our proposition for MCC is the recent emergence of mobile processors from industry dominant microprocessor vendors [2]. They include Intel Mobile Pentium II [3], Intel Celeron [4], and Mobile AMD-K6-2 [5] processors.

The mobile processors run at a lower voltage than desktop processors and operate within the thermal envelope of today's notebook designs. The primary advantage of the lower voltage, lower power mobile processors, is extended system battery life.

These mobile processors are compatible with existing software and offer leading-edge 3D and multimedia performance; in near future, they aim to provide support for high-performance notebook computing.

Although, due to the transmission latency and poor reliability of wireless media, mobile nodes may not be the best choice to participate in a cluster providing a high-performance computation facility for large-scale and grand-challenge applications, there are many applications that will benefit from clustering regardless. Further, this scenario will change as technology becomes mature. In these applications several of its participating nodes may need to be mobile. Examples include oil rig sensors, sensors and monitors in earth quake detection/prediction, marketing representatives travelling all over the country/world with laptops and sales data feeding in, disaster management systems, battlefield command systems, SWAT teams, and stock market wizards. A few sample applications of MCC include the following:

- A scientific, nomadic environment. For example, scientists stationed at several different cities may

[1]Currently associated with the School of Computer Science and Software Engineering at Monash University, Melbourne, Australia.

[2]Cluster is a collection of interconnected computers working together as a single system.

gather seismic data and employ a parallel algorithm that predicts future seismic activities.

- Embedded military supercomputing applications in which fault tolerance is a concern. For example, tanks, trucks, planes, etc. may be connected via a wireless network. They can gather data, send it to other nodes in the network, and use distributed algorithms to make decisions based on that data.
- Intelligent incast or multicast in which strict time deadlines are used to steer immediate actions. For example, consider the study of a weather phenomenon such as tornadoes. Several mobile nodes may gather data, send it to other nodes and use the data to make predictions about the path of the tornado.

Applications of this nature can be executed on a single system, but the data sources are inherently at distant mobile locations and thus necessitate the need for sharing mobile resources. Even in a normal environment mobile computers may be used extensively. The idle CPU cycles of these mobile nodes can be used to process whole or part of the input or output of a large scale application in cooperation with other mobile or stationary nodes. Therefore, mobile cluster computing is expected to play an important role in the modern computing era and mobile network systems.

Furthermore, in case of mobile cluster computing there is no need to invest in a new backbone infrastructure. The existing wireless network infrastructure (such as a cellular network system, PCS, satellite communication system, wireless LAN) can act as a communication backbone for mobile cluster computing. Among several wireless network configurations, a cellular structure is the most commonly used one, where the wireless service area is physically partitioned into different cells. Regardless of whether communication is cellular/stellite-based, they are all based on the wireless media and provide communication environment to the mobile users. Therefore, the infrastructure of mobile cluster computing can be built based on the combination and collaboration of the general cluster computing and the wireless network.

Contribution: In this paper, we define and analyze the application environments for mobile cluster computing. We have raised the potential research issues in mobile cluster computing, among them the timeliness issue in mobile cluster computing, especially during handover events, has been identified and addressed in detail. The contributions of this paper include block-based route optimisation which focuses on timeliness issue in rerouting during handover, and several multicast tree reconstruction algorithms that address the timeliness issue in multicasting during handover.

2 A Mobile Cluster Computer and its Architecture

A cluster generally refers to two or more computers (nodes) connected together with each node having the "strong sense"[3] of cluster membership [1]. A mobile cluster, in addition, consists of mobile and stationary nodes (see Figure 1). The nodes of a mobile cluster communicate either using a wireless network (e.g., cellular network) and/or high speed wired network system. It does not matter where the nodes are and what kind of backbone is used to support the nodes, these nodes can communicate and collaborate with each other just as in a general computing cluster. That is, a mobile cluster consists of both mobile and non-mobile nodes interconnected through either physical interconnect or wireless network and work in a coordinated manner transparently sharing workload among themselves. In addition, a mechanism for automatic detection of node status–whether it is static or mobile–and communication via either physical or wireless network is appropriately supported.

The following are the basic components of mobile cluster computers:

- Multiple High Performance Computers (PCs, Workstations, or SMPs)
- State-of-the-art Operating Systems (Layered or Micro-kernel based)
- High Performance Networks/Switches (such as Gigabit Ethernet and Myrinet)
- Network Interfaces Cards (NICs) (e.g., wireless NIC)
- Wireless Network Infrastructure (e.g., cellular network system)
  - Hardware (such as Base station, MTSO, PSTN)
  - Operating System Kernel (such as mobility management, channel resource management, registration)
- Fast Communication Protocols and Services (such as Active and Fast Messages)
- Mobile Communication Protocols like Mobile IP
- Cluster Middleware (Single System Image (SSI) and System Availability Infrastructure)
  - Hardware (such as Digital Memory Channel, hardware DSM, and SMP techniques)

[3]It referes to the appearance of a collection of independent nodes as a single unified resource and of course, all nodes must aim towards this.

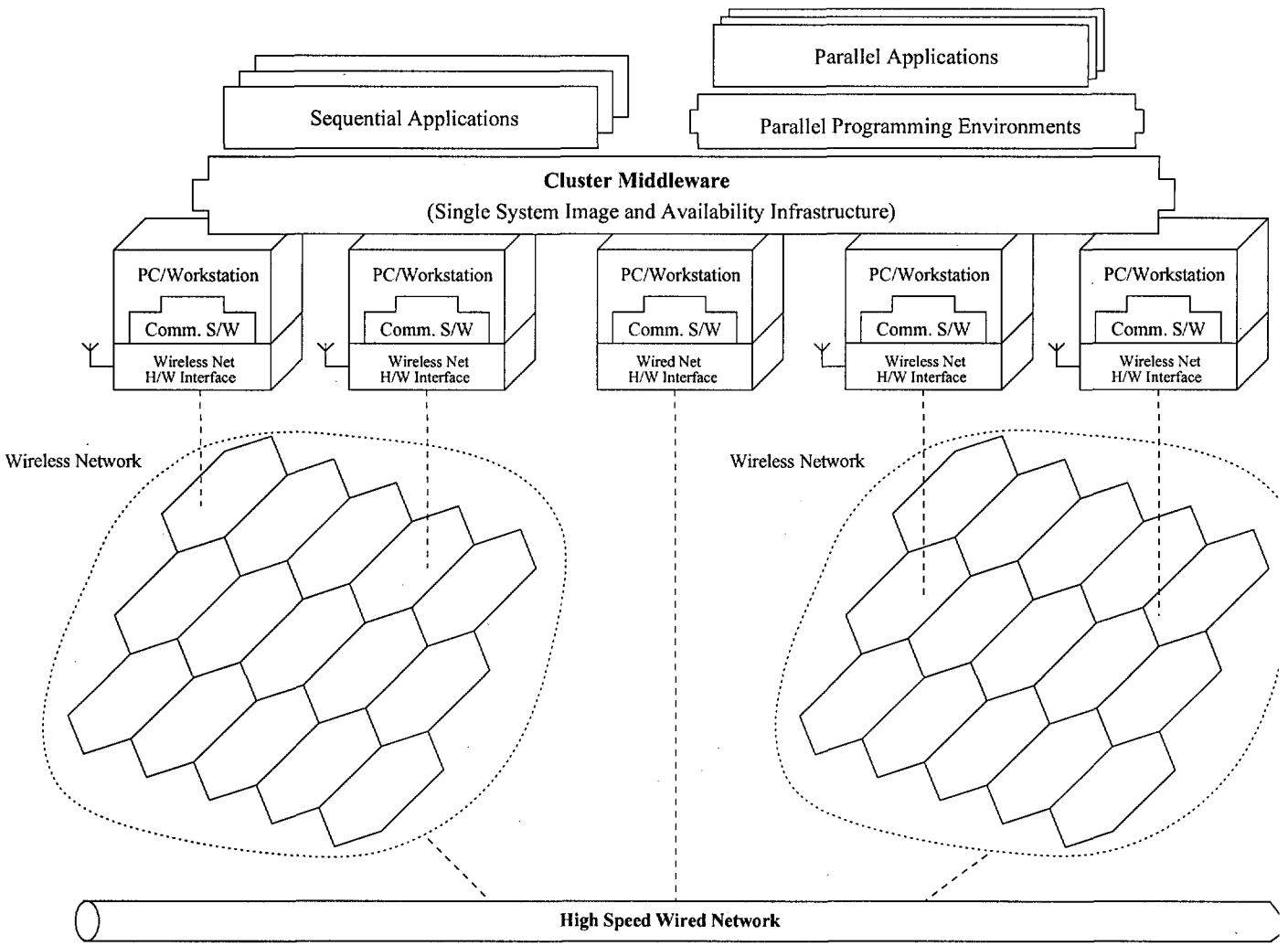


Figure 1: Mobile Cluster Computing Generic Architecture.

- Operating System Kernel or Gluing Layer (such as Solaris MC and GLUnix)
- Applications and Subsystems
  - Applications (such as system management tools and electronic forms)
  - Runtime Systems (such as software DSM and parallel file system)
  - Resource Management Systems (such as LSF and CODINE)
- Parallel Programming Environments and Tools (such as compilers, PVM, and MPI)
- Applications
  - Sequential
  - Parallel or Distributed

The network interface hardware acts as a communication processor and is responsible for transmitting and receiving packets of data between cluster nodes via a network/switch which could be ATM, fast Ethernet, wireless network, etc. When a wireless network is involved to provide communication methods between nodes within a cluster, cluster computing research should consider a lot of issues that the characteristics of mobility in a wireless network bring up. For example, when the mobile node is an Internet node assigned with an IP address, mobile IP, one of the communication protocols, may need to be included to provide a seamless connection across the Internet when the mobile node is roaming.

Communication software offers the means of fast and reliable data communication among cluster nodes and to the outside world. Often, clusters with a special network/switch like Myrinet use communication protocols such as active messages for fast communication among its nodes. They potentially bypass the operating system and thus remove the critical communication overheads providing direct user-level access to the network interface.

The cluster nodes can work collectively, as an integrated computing resource, or they can operate as individual computers. The cluster middleware is responsible for offering an illusion of a unified system image (single system image) and availability out of a collection of independent but interconnected computers.

Programming environments can offer portable, efficient, and easy-to-use tools for development of applications. They include message passing libraries, debuggers, and profilers. It should be noted that clusters could be used for the execution of sequential or parallel applications.

2.1 Infrastructure of Mobile Network

The infrastructure of mobile cluster computing mainly consists of conventional computing clusters and mobile networks. Mobile networks (wireless network along with wired network) provide connectivity and communication methods between nodes within a computing cluster, on which cluster computing relies at the higher layer. There are many implementation technologies of mobile networks as shown in Figure 2.

At the physical and MAC layers in OSI model, a non-exhaustive list of wireless networks is as follows.

- Satellite Communication
- Cellular Networks
- PCS
- Wireless LAN
- Microwave Communication
- High-speed Laser Links

For example, a cellular network system physically partitions the service area of wireless networks into different contiguous cells. Cells have the connotation of geographical area served by a base station. Base station consists of a transmitter and two receivers per channel, a controller, an antenna system, and data links to the cellular office. The base station acts as the user-to-MTSO interface. The Mobile Telephone Switching Office (MTSO) is the physical provider of connections from the cellular radio through the base station to the local exchange carrier. The connection can be on either a landline or a microwave radio system between the points.

Another example is wireless LAN which provides wireless connection for mobile nodes within a small area, such as a building. Currently, a lot of commercial products of wireless LAN have emerged, some of which can reach the transmission rate of 100Mbps. The following list is several examples of some wireless LAN implementation.

- http://www.radiolan.com/
- http://www.wilan.com/
- http://www.ccsinc.net/
- http://www.kcnet.com/dceclan/
- http://www.dataequip.com/

At the network layer, a lot of mechanisms have been proposed or implemented to route the message throughout the entire network system even when the mobile node is roaming around. Such as wireless ATM, dial-up cellular phone connection, mobile IP, etc. For example, the IETF solution to route message for a mobile Internet host is mobile IP. The current research status and some implementation of mobile IP can be found at the following web-sites.

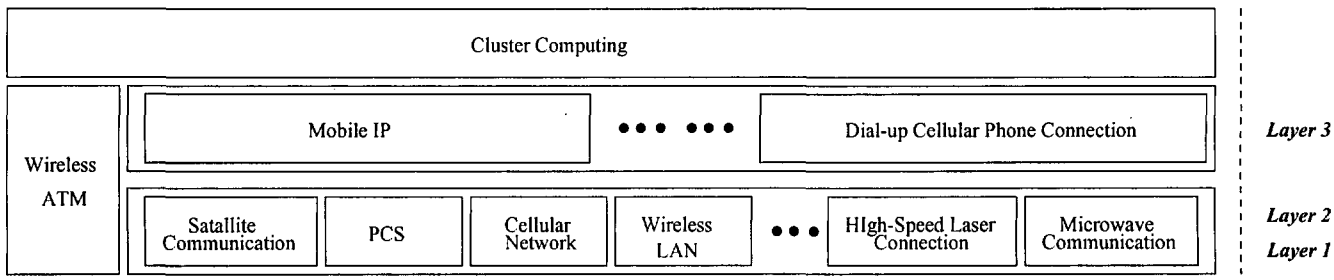


Figure 2: Infrastructure of Mobile Networks.

- National University of Singapore, Mobile IP for Linux - http://mip.ee.nus.sg/
- SUNY, Binghamton - http://anchor.cs.binghamton.edu/~mobileip
- Stanford - http://mosquitonet.Stanford.EDU/software/mip.html
- http://www.cs-ipv6.lancs.ac.uk/MobileIP
- http://www.ikv.de/products/roamin.html

Based on the services provided by the network layer, nodes (no matter mobile or stationary) could communicate and collaborate with each other and process the large-scale computing.

In summary, the wireless network infrastructure provides a communication scheme to every node in a computing cluster in the same way as wired network does. However, due to the introduction of mobility into cluster computing, a lot of new issues come up.

3 Research Issues

Similar to the mobile cellular system, the characteristics of mobility brings up a number of new research topics to mobile cluster computing.

There are a large number of research issues existing in mobile cluster computing, including the following:

- how to balance the long latency of wireless media and high speed fixed network and perform synchronisation;
- how to provide a complete transparency to users without the knowledge of the underlying system which includes both mobile nodes and stationary nodes;
- how to manage the resources and process the scheduling among different nodes to maximise their throughput.

All of these issues are very critical to the performance of mobile cluster computing and further research on these issues is needed. In this paper, we concentrate on the timeliness issue in mobile cluster computing.

One of the most important issues in mobile cluster computing is the timeliness of handovers that occur in the cellular network system. (The set of operations performed when a mobile node moves from one geographical cell to another adjacent cell is called *handover* [6]). When handover takes place, not only the channel resource occupied by the node should be managed to maintain the connectivity, but also the information sent to or from the mobile node should be rerouted to the new cell to which the mobile node is heading. In addition, if a mobile node is within a multicast application when a handover happens, the entire multicast tree [8] may need to be reconstructed. If the time spent on the resource reallocation, rerouting and its corresponding optimisation, and rebuilding the multicast tree is too long, the information sent to or from the migrating node may be lost and this may result in the serious failure of the entire cluster application. Therefore, within a computing cluster, timeliness becomes an important issue when a mobile node switches from one cell to another.

4 Research Focus: Timeliness Issue in Mobile Cluster Computing

Timeliness is a critical issue in many computing and communication applications, especially in real-time systems. A real-time system is one in which the correctness of the system depends not only on the logical results, but also on the time at which those results are produced. Messages transmitted in such systems must be received by a deadline or they are lost. Such a real-time deadline, i.e., timeliness issue, is a key component of the QoS (Quality of Service) requirement. However, even if the system does not have the real-time requirement, timeliness issue is still an important component to provide a high QoS to users. For example, in a TCP/IP network, if an end-to-end acknowledgement doesn't reach the source node (e.g., due to network congestion) before the timer expires, the source node will resend the message which would increase the network load and thus deteriorate the net-

work performance. Therefore, the timeliness issue not only exists in real-time systems but also in a normal system where timing is an important issue.

Similarly, the timeliness of an operation is an important issue in MCC under several circumstances. A few of them are identified and discussed below.

4.1 Resource Management

When a mobile node moves from one cell to another cell, the channel used in the old cell may not be reusable in the new cell due to co-channel or adjacent channel interference or low signal strength leading to node isolation from the rest of the cluster. If a new channel that needs to be used has not been allocated to the node within a short time period, the call may be dropped by the user due to the long waiting-time in a cellular telephone system, or, the messages transmitted will be delayed, resulting in retransmission in a mobile data communication system. If such a delay occurs in a time-critical system, the message may not be able to meet the real-time deadline and could get lost, thus leading to the failure of an entire application [9]. The system resources need to be managed (using checkpointing and migration techniques) to handle sudden unavailability of resources.

4.2 Topology Management

Timeliness is also an important issue in network topology management. Various types of logical topologies have been proposed, such as a *ring topology* where each node has exactly two neighbours, a 2-d mesh [7] which is a planar structure that has nodes arranged in a grid of rows and columns, a *tree topology* which has a regular and hierarchical structure of node levels which creates a tree appearance by having each lower level contain more nodes than the previous level, etc. When a mobile node roams and leaves its original position, the pre-defined topology is destroyed and the fragmented topology has to be reconstructed by selecting a new node as the alternative to the migrated one. The time spent on such reconstruction should not be too long, otherwise neither the cluster computing nor the normal network communication would show satisfactory performance.

4.3 Routing

In a mobile network system with mobile IP protocol, any mobile node is allowed to move about, changing its point of attachment to the Internet, while continuing to be identified by its home IP address. Corresponding nodes sending IP datagrams to a mobile node send them to the mobile node's home address by forcing all datagrams for the mobile node to be routed through its home agent. Thus, datagrams to the mobile node are often routed along paths that are significantly longer than optimal. Route optimisation, an extension of mobile IP, provides a means for nodes to bypass the possibly lengthy route to and from that mobile node's home agent by tunnelling their datagrams directly to the foreign agent. However, when a mobile node rapidly or frequently moves from one cell to its neighbouring cell, route optimisation would be processed too frequently. Since the binding information transfer and the computation of an optimal route at the corresponding node and all intermediate nodes are all time consuming processes, timeliness becomes an important issue in such an environment.

4.4 Multicast

Multicasting is an important paradigm of end-to-end communication, where simultaneous transmission of messages are required from a source to a group of destinations. To route these messages, a multicast tree is required to replicate the data available at the root node of the tree and forward the data along various branches leading to destinations at leaf nodes of the tree. However, in a mobile network system, when handover occurs, the multicast tree would be changed by constructing a new route and deleting the original unused route. Since the time spent on the multicasting tree reconstruction may become significant if many mobile nodes are involved in the multicast tree, the timeliness issue should be considered carefully for each multicast tree reconstruction algorithm.

4.5 Message Interaction Pattern between Different Activities

In a mobile cluster computing system, the nodes in the cluster communicate with each other over networks. Different nodes are responsible for different tasks and have different message interaction patterns. When one or several nodes are waiting for the message sent by a mobile node that happens to have a longer processing time than expected due to handover or some other time consuming activities, the normal operation of these nodes has to be paused until the message from the mobile node becomes available. Such blocking may significantly deteriorate the performance of the entire cluster computing application. Thus the timeliness issue also appears to be an important issue in a message interactive cluster computing environment.

4.6 Synchronisation of Cooperative Activities

In most cluster computing applications, the results from individual tasks distributed among several nodes will eventually be collected and processed by one node, which then generates the final result. If one or more nodes delay the assigned task and do not provide the

result in time due to unexpected events, the final result cannot be generated within a stipulated time. It might lead to failure in a real-time application or reprocessing due to non-synchronisation of the results from other delayed tasks.

5 State-of-the-Art

A lot of research has been conducted in the area of mobile message rerouting based on different system architectures. To successfully reroute a message for the mobile node which is roaming on the Internet, mobile IP [12] is most popularly used. With mobile IP, when a mobile node migrates from one access point to another, the home network of the mobile node will transfer the message between the mobile node and the correspondent node via the foreign network. Thus, for every message meant for the mobile node, it is required to be routed through the home network. This route is efficient when the agent in the home network resides along or near the best route between the correspondent node and the mobile node. Since it is possible for the home agent to be far away from the best or the optimal route between the mobile node and the correspondent node, a route optimisation procedure that would aim to compute a direct and efficient route between the mobile node and the correspondent node is suggested in the route optimisation draft [13] to improve mobile IP. In [14] the operation of mobile computers using IPv6 which enables mobile computers to cache the home network and a care-of address, is specified. The correspondent node then can compute a direct optimum route to the foreign network (care-of address).

A large amount of research has been conducted in real-time multicast. Dijkstra's shortest path algorithm [15] and the Steiner tree generation problem [16] employed for the delay optimisation and the cost optimisation constraints in the real-time multicast can produce traffic and tree height minimised real-time multicast trees. Some heuristics for the Steiner tree problem have been developed that take polynomial time and produce new optimal results [17]. In [18] the KMB algorithm works under the assumption that a network is abstracted to a complete graph consisting of edges that represent the shortest paths between the source node and the destination nodes. The dynamic update of the tree if destination nodes join or leave the tree occasionally is examined in [16]. The dynamic algorithm proposed for the multipoint problem satisfies the bandwidth constraints based on the minimum spanning tree algorithms proposed for the Steiner tree problem. In [19] the optimisation on both traffic cost and delay is discussed; however, the authors assume that the cost and delay functions are identical. A source-based multicast algorithm that can set the variable delay bounds on destinations and can handle variants of network cost optimisation goals is proposed in [20]. Finally, in [21] real-time multicast application which is required to meet specified time and geographical constraints is considered. This study improves steadiness and tightness metrics, defined as functions of maximum and minimum individual point-to-point delay.

6 Timeliness Issue in Routing and Handover

In cellular wireless networks, a mobile node that roams from one cell to a neighbouring cell could be a cellular handset or an IP node assigned with an IP address. No matter what type of mobile node it is, when the handover occurs, the information sent to or from the mobile node should be rerouted in time. However, in current research, for different type of mobile nodes, the mechanism of rerouting and route optimisation is different and thus the corresponding timeliness issue is addressed in a different manner.

6.1 Timeliness Issue in Mobile IP

The basic IETF mobile IP protocol provides for transparent packet routing to mobile hosts on the Internet. This protocol suggests that all the packets must pass through the home agent, which will then tunnel them to the current foreign network. A proposed extension of mobile IP is a route optimisation scheme of sending a binding update message to the correspondent node so that it could perform optimisation in the route to the mobile node. In this approach, the correspondent node is required to maintain a binding cache, which is basically a tuple consisting of the foreign network the mobile node is currently in, the home agent of the mobile node, and the time for which the mobile node will be in the current foreign network. An extension to the route optimisation scheme has also been proposed which suggests an approach where the home network sends information about the mobile node's current location via a piggy-back message. The corresponding node then communicates directly with the mobile node. This approach has the additional advantage that it does not require the binding cache or the binding list to be present.

The current research indicates that there are two different perspectives to a node's mobility. In the first scenario, the mobile node is a station with a fixed IP. The mobile node then moves from the home network to a foreign network. The problem in mobile IP is maintaining connectivity within this mobile node after it moves to the new location (i.e., foreign network) provided it has the same IP as earlier. The mobile IP protocol addresses this problem with the "tunnelling" solution. The Route optimisation based on this is improved by constructing an optimum route

between the correspondent node and the mobile node. In the second scenario, the mobile node is relatively more 'mobile' than it is in first scenario. In a cellular environment, the node is continuously moving between cells, i.e, it is rapidly initiating handovers. In such a situation, an idea has been proposed that alternate routes should be established between the correspondent node and the mobile node so that connectivity could be maintained, even under continuous and rapid handovers.

Consider the integration of the above two scenarios, where the mobile node is an Internet host, and is connected to the Internet via a base station and an onward wired link. The mobile node has a fixed IP address and also rapidly initiates handovers. To effectively maintain the connectivity under such a situation, one would look for a combined solution. Therefore, under this problem scenario, the home network would tunnel the datagram to the foreign network, until such time an alternate route is not established (via the route optimisation philosophy). But when the mobile node is in continuous motion, route optimisation should be performed each time a handover occurs. Since the route optimisation is a time consuming task, it may delay the messages rerouting to the right destination. In addition, if such a route optimisation is performed frequently the network system will get burdened performing this task instead of the normal communication.

To solve this problem, we propose that it is not necessary to perform the route optimisation after each handover, but the optimisation should be based on the motion pattern of the mobile node [22]. Note that the handover here has two connotations: a change in the wireless link and a change in the point of contact. If the binding indicates that the mobile node has an extended period of stay in the foreign network, then the route optimisation needs to be carried out so that the following new messages could be rerouted to the mobile node quickly. However, if the mobile node is to be in the present foreign network for only a short period of time (either because it is continuously in motion or for some other reason), then route optimisation is not necessary because potential handover may happen in a short period of time. So the messages are still forwarded from the home network to the foreign network instead of finding a new route from the corresponding node to the foreign network and then routing all the following messages.

A simulation model has been conducted in [22] that aims to simulate the above circumstance, where mobile node movement has been modelled in four different patterns - linear single dimension movement, completely haphazard movement, straight movement and random movement. It compared the performance of the above two approaches, one of which carried out a route optimisation over every handover, while the other advocates forwarding, where the packet meant for a mobile node in a foreign network via the home network. The simulation results indicate that the overall cost of our approach is less than that of the former approach.

6.2 Timeliness Issue in Cellular ATM System

Like the handover process in mobile IP, efficient handover schemes have been proposed that aim to reduce latency for a cellular ATM network. The virtual connection tree (VCT) concept where virtual circuits are pre-established from the root of the tree to each base station was proposed in [10]. Therefore, a handover involves only the switching between virtual circuits. In [10] it was claimed that admission control is invoked only in new virtual connection establishment and the handover, which is cross to the adjacent virtual connection. Since the geographical area spanned by a virtual connection tree may cover large area and contain many base stations, the frequency of the admission control involvement is still low. Hence, the related handover problems caused by small cells are avoided. On the other hand, [11] doubted the low admission control involvement of the connection tree during a mobile handover, and pointed out that the connection tree generates large overhead during a handover due to call admission processing in every node along the new route. To address this drawback, we proposed a mobile virtual path network architecture (MVPA) where the pre-defined virtual path topology eliminates the need for elaborate call routing functions and switching table. Also, call admission control decisions only need to be executed in the mobile ATM switch (MAS) and the area communication server (ACS).

However, both VCT and MVPA ignore the fact that a large portion of the wireless communication is usually in the same area covered by the same VCT or ACS, which is generally defined as a local traffic. With the VCT or MVPA approaches, all local traffic still has to go through the VCT root or ACS even if the source and destination mobiles are covered by different MASs under the same ACS. The route via the root may not be the shortest path, and is likely to delay the transmission of messages, and also creates a large and unnecessary overhead. As a remedy, [23] proposes a cross-tree concept, namely virtual circuit cross-tree, to separate the local and across tree traffic. Similar to the routing in VCT, the traffic across different virtual connection tree would be sent to the root and transmitted to another virtual connection tree. However, for the local traffic within the region covered by the same virtual connection tree, it will be more efficient to route the traffic from the source node to its nearest parent controller which is able to route the traffic to the destination.

When a handover happens, the mobile node may

move within or across a VCT region. In order to quickly reroute each handover virtual connection to the new cell, a remote and local mobile rerouting path architecture is needed. In this architecture, admission control is not necessary for every handover, but only for the handover across an adjacent region. When local traffic is the major traffic, the latency and overhead resulting from handover could be improved a lot than the original VTC and MVPA approaches.

7 Timeliness Issue in Multicast and Handover

Current trends in networking application, such as multimedia, indicates that there will be an increasing demand in future network for mobile multimedia communication. Multimedia applications require support from the underlying broadband network at the end-to-end communication level. Multicasting is an important paradigm of end-to-end communication. It is a type of group communication which requires simultaneous transmission of messages from a source to a group of destinations. The route of multicast can be viewed as a tree which is a data distribution path consisting of the router nodes and links to carry the data flow from a source to destinations. The routing system replicates the data at the root node of a tree and forwards the data along various branches leading to destinations at the leaf nodes of the tree.

In the cluster computing arena, multicast is also an often used method to distribute message from one node to multiple nodes within a computing cluster. The same principle applies to the mobile cluster computing applications. If a mobile node is a member of a computing cluster as well as a multicast tree which is used to distribute messages among multiple nodes within the cluster, when it handovers from one cell to another cell, the multicast tree may need to be changed by constructing a new route to handover mobile and deleting the original unused route. Since such multicast tree reconstruction is a time consuming process, timeliness becomes an important issue which the network designer should consider. If the reconstruction takes too long, the message sent to the handover node may be delayed or even be lost, then the message received by different nodes will not be synchronised which will lead to the retransmission or pending of some of the processes.

Many cellular networks follow a centralised network management scheme that uses a base station organisation where all communication between nodes is handled by the base stations. Another option for the cellular networks is to use a distributed network management scheme, which is called an ad-hoc cellular network, where several mobile nodes come together in a small area and establish peer-to-peer communication among themselves without the use of the base station. In this section, we discuss timeliness issue in multicast and handover under two types of mobile wireless networks: base-station-oriented and ad-hoc network. The type of nodes constituting the multicast tree in these two networks are different. In base-station-oriented cellular network, the multicast tree is composed by the base stations and the mobile nodes, where the base stations are the intermediate nodes of the tree and the mobiles are the leaf nodes of the tree. However in ad-hoc networks, since there is no base-station, all nodes in the multicast tree are mobile nodes. These nodes connect with each other according to a logical topology and one or many of them may be selected as leader(s) to perform multicast just as a base-station does in the base-station-oriented network.

7.1 Timeliness Issue of Multicast and Handover in Base-Station Oriented Network

Timeliness is a key component of the QoS requirements, and this is commonly measured as end-to-end delay. The purpose of our research is to maintain the end-to-end delay of the multicast session under a tolerable value during handover in a cellular network [24]. We first discuss the timeliness issue of multicast when the handover occurs in a cellular network. Then we commence with a triangle mesh topology. We also propose a new multicast tree reconstruction algorithm under the proposed triangle mesh topology.

7.1.1 Timeliness Issue in Handover and Multicast

When a mobile node handovers from one cell to another cell, the packets should be rerouted from the old base station to the new base station. If the mobile node is also within a multicast tree, when handover occurs, it may no longer receive any multicast messages if the new base station is not within the same tree and will not forward further multicast message to it. Under such circumstance, not only normal rerouting but also multicast tree reconstruction should be accomplished before the handover switching is finished so that the normal communication and multicast session would not be broken. Hence, there are two timeliness aspects we should consider during handover and multicast. One is the time to reroute the messages which has been discussed in the last section, the other is the time for a multicast message to reach the destination which is the focus of this research.

7.1.2 Multicast Tree Reconstruction

Multicast tree reconstruction is applied to adjust the multicast tree and to prepare for the potential handover. It aims to guarantee that the multicast tree can

maintain the timeliness requirement even after been reconstructed. Due to the time cost of reconstructing the tree, we propose to keep a suitable multicast tree which satisfies the timeliness requirement during the multicast session by just adding a hop into the original multicast tree instead of reconstructing a new one after each handover. Such a new multicast tree may not be the optimum tree, but the reconstruction time has been reduced significantly by avoiding the global communication and tree construction (while satisfying the timeliness requirement of multicast). This "just add a hop" multicast tree reconstruction approach is advantageous even when frequent handovers happen to a mobile node, because if during each handover process, the multicast tree should be reconstructed globally, the system resources would be exhausted by a new tree construction instead of the normal communication. But when such a non-optimal tree cannot satisfy the timeliness requirement of multicast, a new multicast tree is required to be constructed. The process of deciding when to reconstruct the tree is called *multicast tree optimisation condition judgement.* This judgement assumes the direction of the next handover, estimates the end-to-end delay of the new path by adding a hop to the original path, and then compares such delay with the timeliness requirement. If the time delay along the new path is still within the tolerance of the normal multicast requirement, the new multicast tree construction should be initiated.

7.1.3 Triangle Mesh

Before the introduction of the new multicast tree construction and reconstruction algorithm, we commence with a new cellular network topology. In cellular network, hierarchical topology is the most commonly used model where PSTN, central controllers, base stations and mobile users construct a hierarchical architecture. All the traffic across the inter or intra-region cells will pass through the central controllers; this makes the cellular network system constructed as a multiple layer star topology with the central controller acting as the star centre. This topology is indeed a centralised topology where the central controller may become the bottleneck during a heavy loaded traffic hour. To solve this problem, we commence with a new topology where each base station is connected with other base stations in its neighbouring cells. In this way, the intra-region traffic are processed by base stations themselves through the physical links while the central controllers only handle the inter-region traffic. If we idealise every cell as a regular hexagon, the entire connection among the base stations is a triangle mesh. We also retain the hierarchical architecture at a higher level as it used to be.

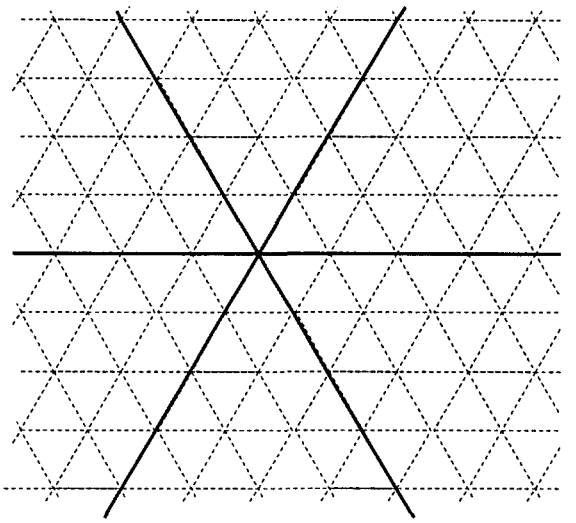


Figure 3: Triangle Mesh.

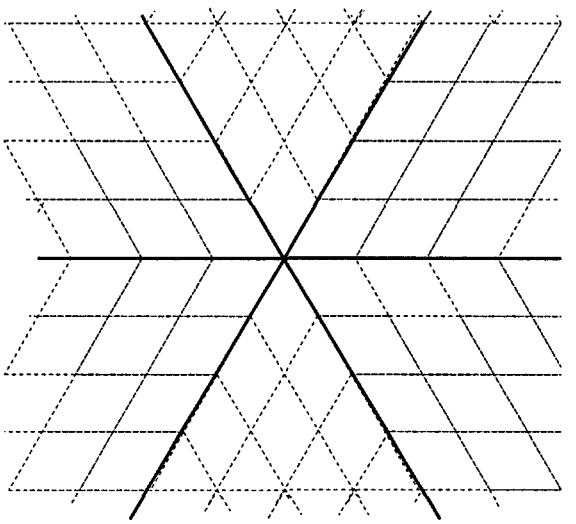


Figure 4: Optimised Triangle Mesh.

7.1.4 Multicast Tree Construction and Reconstruction in Triangle Mesh

In the cellular network with the above topology which combines hierarchical architecture and triangle mesh architecture together, there are two types of handover - intra-region handover and inter-region handover. We focus on the intra-region handover in this research, while our algorithm over triangle mesh can be easily extended to the whole network.

As shown in Figure 3, a triangle mesh can be divided into the six areas along three lines across a source node. Since each line in a sparse direction will not be the part of the shortest path from the source node to the destination node, we can erase all the lines in the sparse direction inside every area. This is shown in Figure 4, where the triangle mesh changes to quarter 2D mesh in each of the six areas. To construct the multicast tree in this quarter 2D mesh, we distribute all the nodes

into six areas and get the common least ancestor node of every area. Then inside every area, we construct a submulticast tree which has the root at the common least ancestor node by using any of the existing 2D mesh multicast tree construction algorithms. With such a construction, we can get a multicast tree on a triangle mesh, which satisfies the condition that the path from the source node to every destination is the shortest path.

When a mobile node handovers across cells, and "just add a hop" approach is not able to satisfy the timeliness requirement, multicast tree should be reconstructed. First, we get the node that needs to be added to the tree (i.e., the new base station in the cell that the mobile is moving to) after last tree construction or reconstruction. Second, we get the nodes that are not necessary to be included in the tree any more because the mobiles inside these cells have already left for other cells. The reason that they are still in the tree is the "just add a hop" approach includes them as temporary. When the whole tree is being reconstructed, it is not necessary to include these nodes in the tree. If these nodes are leaf nodes, we remove them from the tree and also disconnect the links connecting them from other nodes. Then we check their parent nodes applying the same method and repeat the process until there is no node that is not necessary to be included into the new multicast tree. Finally, for every node that needs to be included into the new tree, we calculate the shortest path from all the nodes inside the tree to it, and use the shortest one that can satisfy the end-to-end delay requirement for each node. Then we add the node into the tree and set up the corresponding links. The multicast tree is then reconstructed.

7.2 Timeliness Issue of Multicast and Handover in Ad-hoc Network

In ad-hoc cellular networks, the timeliness issue becomes more important and needs serious consideration because the multicast tree is composed of mobile nodes which may handover from one cell to another cell at any time. When handover happens, the nodes connected to the migrating node become orphans and the multicast tree becomes disconnected. If the tree cannot be rebuilt soon, multicast message would not reach all the destinations within the time period it requests. We consider the multicast tree reconfiguration problem on an ad-hoc cellular network. Our purpose is not to generate the 'best' multicast tree, but to reconstruct a disconnected one [25].

For ease of multicast operation, rapid maintainability and other well known advantages of symmetric topologies, we embed the multicast tree on the top of a mesh of hypercubes topology for the intra- and inter-cell networks. In this logical topology, nodes within each cell are connected as a binary hypercube, while the nodes across the cells are connected as a hexagonal mesh. Heuristics multicast algorithms for hypercubes are proposed in [26], and anyone of them can be used to construct the initial multicast tree. We focus our discussion on the timeliness issue of the multicast tree reconstruction when the handover happens to a mobile node.

In ad-hoc cellular network, an orphan node is created when its parent node migrates. A node can also become an orphan, if it migrates and it is a destination node in the multicast tree. We identify three possible situations for creation of the orphan nodes.

- First, the migratory node is a lead node. Since this node migrates to a new cell, it becomes an orphan and it is necessary to find a new parent node belonging to the multicast tree.
- Second, the migratory node is an intermediate node, but not a multicast destination. In this case, the migratory node itself does not need to re-connect to the multicast tree. But, its children nodes become orphans and are required to find new parents.
- Third, the migratory node is an intermediate node, and is also a multicast destination. In this case, the migratory node needs to find a new parent after moving into the new cell, and at the same time its children become orphans and each one of them is required to find a new parent node.

Regardless of whether the node became an orphan due to its own migration, or its former parent node's migration, the net effect is identical, i.e., the fact that it is required to find a new parent. To accomplish that, a rapid replacement of the migrated node into the hypercube topology is first needed, and then a rapid reconfiguration of the multicast tree is required.

To keep the maximum distance from the root to any one of the leaf nodes minimised, it is preferred to create a balanced tree, instead of a skewed one. In doing so, if the multicast tree is uniformly spread and balanced to begin with, then during each tree reconstruction phase, if the orphan node(s) can connect to a new parent belonging to a tree at the same level as that of the previous parent, then the balanced property of the tree remains unchanged. Otherwise, the tree may become progressively imbalanced. As the tree gets more and more imbalanced, some of the leaf nodes will get unduly away from the root, causing additional message transmission delay. Hence our design objective is to have an orphan node get a parent whose tree level is most similar to that of the node's former parent. We propose two approaches below for rapid multicast tree reconstruction based on the proposed hypercube topology.

7.2.1 RRR Approach

The first solution approach, named "Request-Reply-Rejection" (RRR) is designed to operate at run-time, i.e, to be invoked at a time the multicast tree gets fragmented (due to node migration) and is required to be connected or reconstructed. With this approach, when a node migrates, an orphan node at the l-th level will attempt to find a parent so that orphan node request other nodes to be their parent. The node receiving the request either accepts or rejects the request depending on whether or not the timeliness constraints of message transmission can be satisfied. Also the node requested and accepting to be the new parent must be a hypercube adjacent to the requesting one (i.e., the orphan). This RRR algorithm can also be used to improve the multicast tree. During the system idle time, any node connected to a parent may use the same approach to find a more optimal parent, and consequently update the multicast tree. To maintain a "good shape" multicast tree, and avoid creating a skewed shaped one, ideally, each reconstruction step should maintain a height-balanced tree, where the difference between the maximum and minimum height of the leaf is at most one. However, including this check in each step of the RRR algorithm can make the algorithm computationally expensive. Besides, there is no guarantee that among the available set of topologically adjacent parents of the orphan node at least one would offer height-balance maintaining connection. Our proposed approach is to first list all the available parent nodes and their topological adjacency. Next, these available parent nodes are sorted in the tree level discrepancy from the previous parent of the orphan node. Clearly, the first element of the sorted list would offer a parent node that is closest in tree height level to the former parent of the orphan node.

7.2.2 LAP Approach

Another approach, named "Look-ahead Alternate Parent" (LAP) is based on precomputed alternate multicast tree reconstruction techniques. It is required to execute the RRR algorithm or an equivalent one in the background. In the LAP algorithm, each node has a local parent table containing the current parent and a number of alternate parents computed in the background. Due to one or more node migrations, when orphan node(s) are created, each orphan node readily selects one alternate parent from its alternate-parent table. Ideally, this approach would be able to reconstruct the multicast tree in zero waiting time; however, in practice a small table lookup delay would be involved. Also, if the alternate parent table is not ready and available, then associated delays for executing RRR algorithm in the background would also account as a foreground delay.

8 Conclusion

We have discussed the architecture of mobile cluster computer, timeliness issue of rerouting, and multicast when handover occurs in the mobile cluster computing area. Several solution approaches based on different system architectures have been discussed. To successfully reroute the message from the home network to the corresponding node when a mobile node handover occurs a mobile IP tunnels the message from the home network to the foreign network until route optimisation is accomplished. To accommodate the situation when mobile node handovers occur frequently and rapidly, we propose the idea to delay the route optimisation to a certain period and tunnel the message from the home network to a new foreign network so that the the system would not get burdened by frequent route optimisation. In an ATM-based cellular network, the messages are rerouted from the base station of the mobile node to the central controller, which makes the central controller to become the bottleneck and thus the timeliness requirement may not be meet. We propose that base stations within the same region communicate with each other and process the intra-region rerouting by themselves. For the timeliness issue in multicast during handover, we propose the "just add a hop" idea based on a base-station oriented triangle mesh topology so that the multicast tree construction and reconstruction become more efficient. In addition, in an ad-hoc cellular network, we commence with a hypercube topology where two approaches–RRR and LAP–are proposed to accelerate the multicast tree reconstruction.

With rapid developments/progress in the area of cluster computing, reliable wireless communication, mobile computing, and availability of applications that exploit this integrated infrastructure, mobile cluster computing is poised to become a reality in the coming 21st century.

Acknowledgments

We thank Toni Cortes (Universitat Politècnica de Catalunya, Spain), Dan Hyde (Bucknell University, USA), Lars Rzymianowicz (University of Mannheim, Germany), Norm Matloff (University of California, Davis, USA), Alessandro Bevilacquai (INFN Istituto Nazionale di Fisica Nucleare, Italy), Alfred C. Weaver (University of Virginia, USA), and Harjinder Sandhu (York University, Canada) for their suggestions and comments.

References

[1] M. Baker and R. Buyya, "Cluster Computing at a Glance", *High Performance Cluster Computing: Archi-*

tectures and Systems, Vol. 1, 1/e, Prentice Hall, NJ, USA, 1999.

[2] Intel, "Mobile Computing", http://www.intel.com/mobile/

[3] Intel, "Mobile Pentium II Processors", http://www.intel.com/mobile/pentiumii/

[4] Intel, "Mobile Celeron Processor", http://www.intel.com/mobile/celeron/

[5] AMD. "Mobile Processors", http://www.amd.com/products/cpg/mobile.html

[6] A. Noerpel and Y. Lin, "Handover Management for a PCS Network", *IEEE Communications Magazine*, Vol. 3, No. 6, pp. 18-24, Dec. 1997.

[7] P. McKinley, H Xu, A-H. Esfahanian and L. Ni, "Unicast-based Multicast Communication in Wormhole-routed Networks", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, pp. 1252-1265, Dec. 1994.

[8] S. Bhattacharya, C. Albert, and W. Tsai, "Fault-tolerant Multicast on Hypercubes", *Journal of Parallel and Distributed Computing*, Vol. 23, pp. 418-428, 1994.

[9] H. Zheng and S. Bhattacharya, "Working Set in Channel Management in Cellular Networks", *Proceedings of ISADS'99 Conference*, Mar. 1999.

[10] Acampora and M. Naghishineh, "An Architecture and Methodology for Mobile Executed Handoff in Cellular ATM Networks", *IEEE JSAC*, Oct. 1994, pp. 1365-1375.

[11] H. Vogel, "A Networking Concept for Wide Area Mobility and Fast Handoff in ATM Networks ", *IEEE Globecom*, v2. 1997, pp.1124-1128.

[12] B. Lancki, A. Dixit, and V. Gupta, "Mobile IP: Supporting Transparent Host Migration on the Internet", *Linux Journal*, Aug. 1996.

[13] C. Perkins and D. Johnson, "Route Optimisation in Mobile IP", *Internet draft*, Nov. 1997.

[14] D. Johnson and C. Perkins, "Mobility Support in IPv6", *IETF mobile IP Working Group*, Mar. 1998.

[15] E. Dijkstra, "A Note on Two Problems in Connection with Graphs", *Numerische Mathematik*, Vol. 1, pp. 269-271, 1959.

[16] B. Waxman, "Delay-bounded Steiner Tree Algorithm for Performance-Driven Layout", *Journal on Selected Areas in Communications*, Vol. 6, 1617-1622, Dec. 1988.

[17] H. Takahashi and A. Matsuyama, "An Approximate Solution for the Steiner Problem in Graphs", *Mathematica Japonica*, Vol. 6, pp. 573-577, 1990.

[18] L. Kou, G. Markowsky, and L. Berman, "A Fast Algorithm for Steiner Trees", *Acta Information*, Vol. 15, pp. 141-145, 1981.

[19] K. Bharatkumar and J. Jaffe, "Routing to Multiple Destinations in Computer Networks", *IEEE Trans. on Communications*, Vol. COM-31, pp. 343-351, 1983.

[20] Z. Qing, M. Parsa, and J. Garcia-Luna-Aceves, "A Source-based Algorithm for Delay Constrained Minimum-cost Multicast", *Infocom'95*, 1995.

[21] M. Pokam and G. Michel, "Guaranteeing Spatial Coherence in Real-time Multicasting", *International Symposium on Information Theory*, pp. 41, May 1995.

[22] S. Palangala and S. Bhattacharya, "Finding When Route Optimisation Is Necessary", *Technical Report*, CSE Dept. of Arizona State University, Dec. 1998.

[23] C. Yu and S. Bhattacharya, "Message Delivery Delay Refinement of Large Scale Dynamic Networks", *Technical Report*, CSE Dept. of Arizona State University, Dec. 1998.

[24] Y. Chen, C. Yu, and S. Bhattacharya, "Real-time Multicast Reconstruction of Handover Process on Cellular Network", *Technical Report*, CSE Dept. of Arizona State University, Oct. 1998.

[25] B. Gannod and S. Bhattacharya, "Real-time Multicast in Wireless Communication", *Real-time Special Issue of Parallel and Distributed Computing Practices Journal*, Vol.2 (1), 1999.

[26] Lan Y., Esfahanian A-H., and Ni L.M., "Multicast in Hypercube Multiprocessors", *Journal of Parallel and Distributed Computing*, Vol. 8, 1990.

High-Performance Cluster Computing over Gigabit/Fast Ethernet

Janche Sang
Department of Computer and Information Science, Cleveland State University, Cleveland, OH 44115
Phone: 216 6874780, Fax: 216 6875448
Email: sang@cis.csuohio.edu

Chan M. Kim, Thaddeus J. Kollar, and Isaac Lopez
NASA Lewis Research Center, Cleveland, OH 44135
E-mail: {Chan.M.Kim, Thaddeus.J.Kollar, Isaac.Lopez}@lerc.nasa.gov

Keywords: Cluster Computing, Gigabit Ethernet, Parallel Computation, Performance Measurement

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 1, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

Clusters of workstations are often considered to be an attractive platform for low-cost supercomputing, especially if a high-speed network is used to interconnect high-end workstations. In this paper, we investigate distributed network computing on a tree-structure cluster which consists of intermediate and leaf workstations. The intermediate workstations are interconnected together by a Gigabit Ethernet full-duplex repeater and can be used as a gigabit cluster testbed. The leaf workstations are connected to intermediate workstations via Fast Ethernet and form a 100Mbps cluster testbed. We study the performance characteristics involving end-to-end communication and collective communication. The performance of the 100Mbps cluster and the gigabit cluster are empirically evaluated by using a mobile-thread based parallel simulation application and the NAS Parallel Benchmarks. We also discuss the factors which may affect the performance of cluster computing.

1 Introduction

Recently the distributed network computing environment has become a cost-effective computing infrastructure because it provides aggregate resources of computational power, communication, and storage[11]. This environment usually consists of a cluster of workstations which are connected to a Local Area Network (LAN) such as Ethernet for information exchange and coordination among the processors.

Ethernet, also known as a CSMA/CD network, is the most widely used type of LAN. Originally evolved from a link of coaxial cable, it is usually implemented as a 10 Mbps twisted-pair cable network wired with hubs. Though the power and performance of desktop computers have grown tremendously, Ethernet could still provide a surplus of bandwidth in the late 1980s. However, in the early 1990s, with more and more data-intensive and communication-oriented applications, the demand for more network bandwidth was reaching a critical stage. In 1995, Fast Ethernet, an updated standard based on the previous 10 BaseT network, was derived to provide a tenfold increase in performance.

The relationship between computing and communication technology is like a swinging pendulum. That is, the performance bottleneck may be either in the communication systems or in the computers when time changes. Recently, the increase in microprocessor speeds has exceeded the capability of Fast Ethernet connections. To meet the demand for greater network capacity, the Gigabit Ethernet standard was completed and approved in 1998. It adapts existing technology by layering the well-understood and well-characterized IEEE 802.3 MAC on top of the already developed and tested physical layer of the ANSI standard Fiber-Channel with an 8B/10B block coding system. Table 1 briefly compares the technology between the Fast Ethernet and the Gigabit Ethernet. A detailed description about Gigabit Ethernet and Fast Ethernet can be found in [15] and in [7], respectively.

The Advanced Computational Concepts Laboratory (ACCL), located at NASA Lewis Research Center, seeks low-cost high-performance solutions to analyze data for NASA aerophysics applications. The Gigabit Ethernet was chosen by ACCL to fulfill the needs of a cluster environment for its high throughput and simple deployment. We utilized a two-level tree-structure network topology to construct the parallel testbed. The two-level cluster consists of intermediate and leaf workstations. The intermediate workstations are connected together by a Gigabit Ethernet full-duplex repeater and can be used alone as a gigabit cluster testbed. The leaf workstations are connected to in-

Network Type	Data Rate	Wire Type	Coding	Slot Time	Interframe Gap
Fast Ethernet (100Base-X)	100 Mbps	STP, Single-mode and multimode fiber	4B/5B	512 bit-time (5.12 μs)	96 bit-time (960 ns)
Gigabit Ethernet (1000Base-X)	1000 Mbps	STP, Cat. 5 UTP, Multimode fiber	8B/10B	4096 bit-time (4.096 μs)	96 bit-time (96 ns)

Table 1: Fast Ethernet vs. Gigabit Ethernet

termediate workstations via Fast Ethernet and form a 100Mbps cluster testbed.

The objectives of this research were to determine the performance characteristics and associated overheads of Gigabit/Fast Ethernet LANs, to empirically examine the performance differences between the gigabit cluster testbed and the 100Mbps cluster testbed, and to study the factors which may affect the performance on the cluster testbeds. We first measured the end-to-end communication latency and throughput. We observed that a high end workstation (such as Pentium II 400MHZ) can saturate a 100Mbps LAN, but can only generate 20% of the traffic of a Gigabit Ethernet. Then, we used a message passing library based on the MPI standard to measure the performance of collective communication on different cluster testbeds. Collective communication operations, such as barrier, scatter, gather, broadcast, etc., are the major components of message-passing based parallel programs. These operations allow a group of processes running on different computers to synchronize and exchange data by invoking the same function with matching arguments.

We also conducted experiments by running distributed applications on the two experimental platforms. For computational intensive applications, performance can be improved through a good compiler with efficient code optimization capability. For communication intensive applications, in addition to the network speeds, matching the collective communication structure with the cluster interconnection network topology can greatly reduce communication overheads and hence improve performance.

The remainder of the paper is organized as follows. In Section 2, we describe the configuration of the testbed system. Section 3 presents the end-to-end and collective communication characteristics. In Section 4 we extensively evaluate the performance of the clusters. A brief conclusion is presented in Section 5.

2 System Environment

A two-level tree-structure network topology is used to construct the cluster testbed. The tree cluster consists of a gigabit full-duplex buffered repeater *PacketEngines*© $FDR12^{TM}$, a server with 9GB disk, eight intermediate workstations (400MHz 128MB Intel P6, named page01 - page08), and thirty-two dual-CPU end-nodes (400MHz 512MB Intel P6, named grunt01 - grunt32). The $FDR12^{TM}$ connects the server and the intermediate workstations via *PacketEngines*© $G-NIC^{TM}$ interface cards at 1000Mbps rate. Each of these eight intermediate workstations branches out to connect four leaf workstations with a four-port *Adaptec*© $ANA^{TM}-6944A$ Fast Ethernet adapter. Figure 1 depicts the network topology of the P6 cluster. The advantage of the tree-structure setting is that it provides us two cluster-computing testbeds: one is the gigabit cluster of the page machines and the other is the 100 megabit cluster of the grunt workstations.

The $FDR12^{TM}$ is a buffered repeater which merges switched and shared design concepts[10]. Therefore, it can achieve switch-like performance while maintaining the lower cost of a shared hub. The $FDR12^{TM}$ provides 12 ports and each of which is full-duplex and collision-free. When a frame arrives at a port, it will be buffered (like a switch) and forwarded to every other port without address filtering (like a hub). If several frames are received/buffered at the same time on different ports, the round-robin arbitration mechanism is used to ensure fair allocation of bandwidth among ports. Since it is full-duplex, the ideal aggregate bandwidth can be up to 2 gigabits per second.

Both of the eight intermediate workstations (named page01 - page08) and thirty-two leaf workstations (named grunt01 - grunt32) are running the LINUX operating system. There are two possible ways to set up the intermediate workstations to forward packets between leaf nodes. One is configured as bridges and the other is set up as routers. It was decided to try bridging first because bridging functions are implemented in the data link layer which is more efficient and more versatile than the network layer where routers are located. Our experience of setting up the intermediate workstations as bridges was unsuccessful. This is because the maximum size of a packet in Gigabit Ethenet is larger than the maximum size in Fast Ethernet and hence packet fragmentation is necessary if a very large packet sent from Gigabit Ethenet to the Fast Ethernet. Unfortunately, a recent version LINUX bridge by itself couldn't handle packet fragmentation. Currently, we successfully set up the intermediate workstations to support IP forwarding capability, though the router and servers would all have to keep big routing tables.

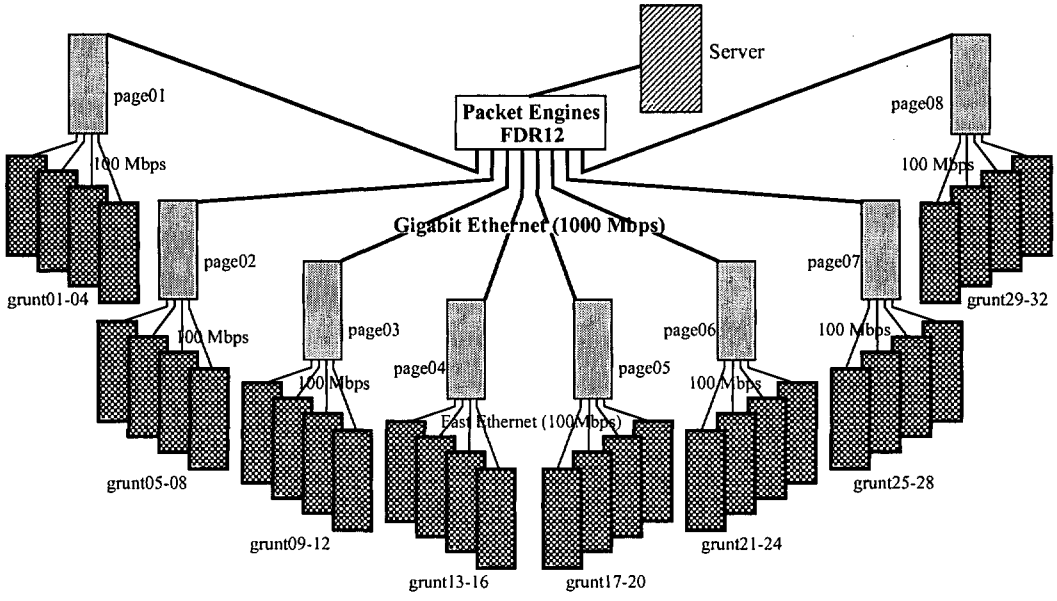


Figure 1: Network Topology of the P6 Cluster

3 Communication Characteristics

3.1 Latency

To measure end-to-end message passing performance, we used the ping command which is based on the ICMP protocol. This command sends an ICMP ECHO_REQUEST message of a fixed size from one node to another and then waits for the response back. The message size was varied from 1 byte to 40K bytes, For each message size, we repeated the experimentse 30 times and then calculated the average of the round-trip latency.

We chose three different pairs of nodes which represented possible message passing paths in the tree cluster. The first pair was from page01 to page02 via the FDR. The second pair was from grunt01 to grunt02 via the intermediate node page01. The third pair was from grunt01 to grunt05 by way of page01, FDR, and page02. Figure 2 illustrates the round-trip message passing times for these three different pairs. The third pair required the longest time because it took 4 hops to reach the other end. The first pair was the fastest one because it is a direct link via Gigabit Ethernet. It can be observed that the summation of the latenceies of the first and the second pairs are roughly equal to the latency of the third one.

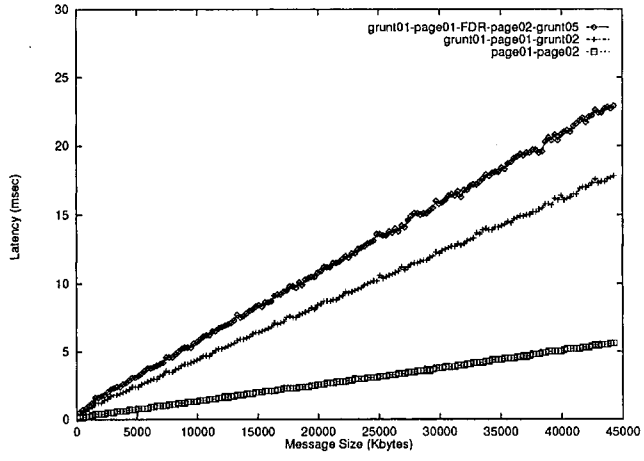


Figure 2: Round-trip Latency

3.2 Throughput

We used two hosts, one sender and one receiver, to estimate the maximum throughput over the link. The sender transmits a TCP/IP message across the link to the receiver, as fast as it can. The receiver uses the function `gettimeofday()` to measure the time to receive the message. The two hosts exchanged initial synchronization messages using the three-way handshake protocol to ensure the connection was established before timing operations began. This technique has been widely used in network throughput measurement[2, 5, 8, 13]. We used TCP because it can provide reliable transmission.

We first chose two end workstations (e.g. grunt01 and grunt02, or grunt01 and grunt05) and found that the maximum TCP throughput could be up to 94 Mbps. This is because the bottleneck in the communication path grunt01-page01-grunt02 (or grunt01-page01-FDR-page02-grunt05) is between the page and grunt machines (i.e. 100Mbps) .

We then selected two intermediate workstations to be the sender and the receiver (e.g. page01 and

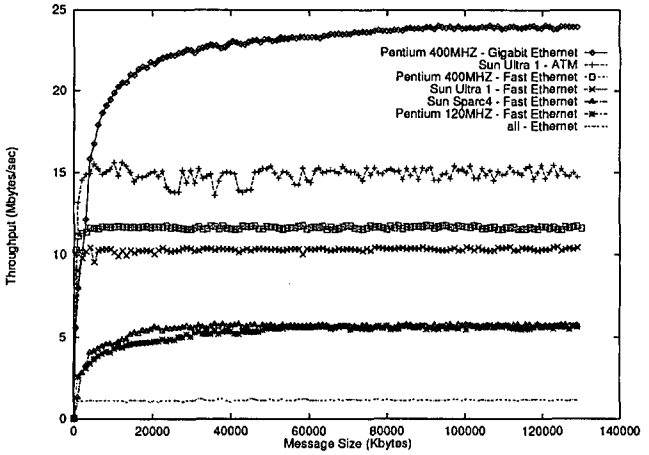


Figure 3: TCP Throughput over Ethernet family

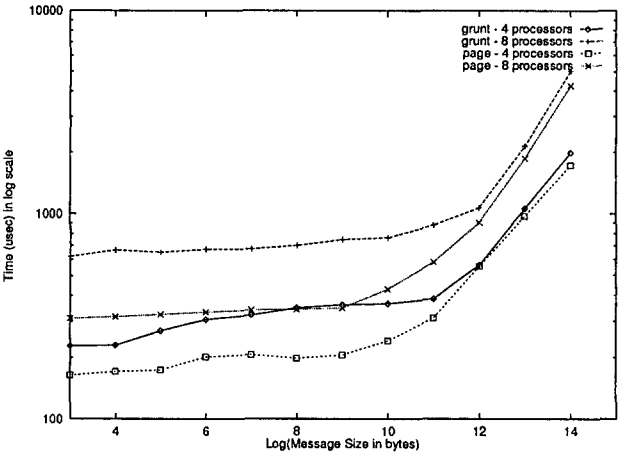


Figure 4: Performance of MPI_Scatter

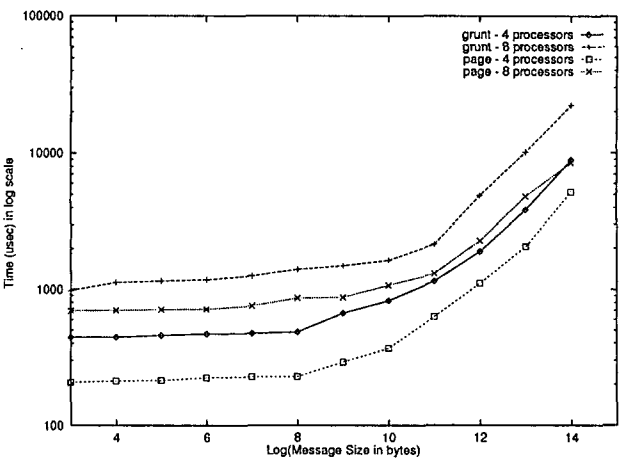


Figure 5: Performance of MPI_Gather

page02). We found that the maximum throughput is only 192Mbps which is only 19.2% of the total 1Gbps bandwidth. This means that even the high-end 400MHz P6 workstation is not fast enough to saturate the network traffic due to some protocol stack overheads such as copying the data between user and kernel address spaces, transferring data from memory to interface, etc.

Similar empirical results can also be found in [5], [8] and [12] which showed that the slower-speed workstations such as SPARC4 and SPARC5 can not produce traffic to saturate the 155 Mbps ATM network. Our early results presented in [12] showed that faster-speed Ultra1 stations can saturate fast Ethernet and generate up to 125 Mbps in an ATM network. Like a pendulum swinging, at any given the computing speed may outpace that of the communications infrastructure, or vice versa. For example, an early paper which measured the Ethernet performance showed that using VAX780 or Sun II the maximum throughput was only around 750K bits/sec , much less than 10 Mbits/sec [2].

We also used four pairs of stations to send/receive packets simultaneously and then calculated the total throughput by summing up the result from each pair. We found that the aggregate throughput was 770Mbps. If each station ran both sender and receiver programs, the throughput could be up to 954Mbps which was very close to the 1000Mbps maximum bandwidth.

3.3 MPI Collective Communication

MPI (Message Passing Interface) is a standard specification for writing portable message-passing parallel programs. It features a range of functionality, including point-to-point, with synchronous and asynchronous communication modes, and collective communication such as barrier, scatter, gather, broadcast, etc. A collective operation is executed by having all processes in the group invoke the common communication function, with matching arguments. There are multiple implementations of MPI. We used MPICH [6] which is developed by Argonne National Laboratory and Mississipe State University because of its wide availability. MPICH was built using the P4 device layer, so all communication was performed on top of TCP sockets. Parallel programs can be started by the *mpirun* front-end shell script which takes the name of the program and the numbers/names of the processing nodes (via the options `-np` and `-machinefile`) to use and then remotely spawns the processes on the cluster.

Collective communication operations are commonly used in parallel programs. Their performance is of great interest. For example, earler study in [9] made a comparison of the collective communication performance between two programming environments. We chose the following test cases as the benchmarks:

- **MPI_Scatter** distributes the ith block of an n-block array, (n is the number of processors in the processor group) to the ith processor in the

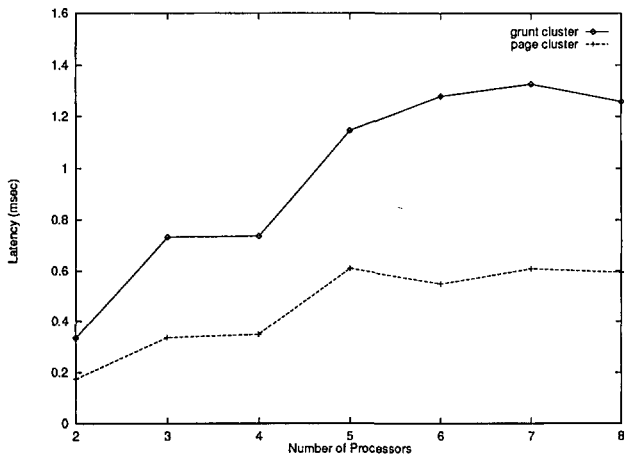


Figure 6: MPI_Barrier Synchronization Time

group.

- **MPI_Gather** reverses the function of MPI_Scatter by collecting all of the n blocks from the other processors in the group.
- **MPI_Barrier** blocks the caller until all processes in the group have called.

Figure 4 and Figure 5 shows the time of MPI_Scatter and MPI_Gather, respectively, on four and eight workstations as a a function of the message size. For the comparison purpose, the experiments were performed on both of the grunt and the page clusters. Since the MPICH collective communication operations are implemented on top of the point-to-point layer, it can be observed that the communication times grow significantly for large message sizes (i.e. ¿ 1K bytes) when they are doubled the size.

We also measured the performance of the MPI_Barrier synchronization time by varying the number of processors in the processor group. The result is depicted in Figure 6. It can be seen that the synchronization times are in the same level when the number of processors varied from 5 to 8. It is also true when using 3 processors and 4 processors. That is, the synchronization time t and the number of processors n have the following relationship:

$$t = c * \lceil log(n) \rceil$$

where c is a constant.

This is because the MPICH implements the barrier function using the "power-of-two"-based algorithm which can provide good scalability.

4 Performance Measurements of Distributed Applications

We used a parallel simulation application and the NAS Parallel Benchmarks to evaluate the scalability/performance of the clusters. We also studied the factors which may affect the performance of cluster computing.

4.1 Mobile-Thread based Parallel Simulation

A parallel discrete event simulation consists of collection of cooperating LPs, each representing one or a set of physical processes(PPs)[4],. Each LP has its own local simulation clock, an event calendar, and input and output communication channels for interaction with other LPs. In our model, an LP is also a host to a set of lightweight processes (i.e. threads), and shared objects (e.g. facilities), as shown in Figure 7. Processes are used to model active components of a system. In contrast, facilities are objects used to model passive system components with mutually exclusive access. That is, processes are dynamic entities which can request access to static facility entities, use these facilities for a time period, and eventually release them to proceed with different activities.

To represent our model formally, assume that an LP hosts processes $p_1, p_2, \cdots$, and p_n. Let t_{p_i} denote the reactivation time of the process p_i and T' represent the time of the next process to be executed in this LP. That is, $T' = \min_i\{t_{p_i}\}$. To guarantee a correct execution of the simulation, each LP must satisfy the causality constraint: events executed by an LP are in nonincreasing temporal order. Note that if a conservative algorithm[3] is used, the next process can be reactivated if the time T' is no greater than the time $\min_k\{t_{c_k}\}$ where t_{c_k} are the clock values of the incoming channels. In other words, each LP may execute a next event e from its list of candidate events only after it is guaranteed that it will not receive an event with timestamp smaller than the timestamp of event e from any other LP. This can be accomplished through the use of a *lookahead* mechanism.

To parallelize process-oriented simulations, our approach was to distribute passive objects across processors, guaranteeing all processes (i.e. threads) easy access to these objects. Naturally, a problem arises when a process executing on some processor requires access to an object that is not located on the same processor. For example, a process hosted by processor A may require access to a resource or facility object situated on processor B. Our solution is to move the requesting process, along with its simulation timestamp, to the site on which the passive object is located. Consider the situation where a process on some host attempts to make consecutive access requests to an object on some remote host. Such a situation can be seen in Figure 7, where we assume that customer C_0 makes a series of access requests to facilities $F_n, \cdots, F_{2n-1}$ located on a remote host after it leaves facility F_{n-1}. Migrating the process to the remote host will reduce the cost of

No. of procs	1	2	4	8	16	32
8x64	425	283	167	88	46	25
(Speedup)	1	1.5	2.6	4.9	9.3	17.1
16x256	3372	1984	994	505	258	133
(Speedup)	1	1.7	3.4	6.7	13.1	25.4

Table 2: Times (in seconds) for simulating closed queueing network on the grunt cluster

communication since the series of access requests will now be made locally instead of remotely.

We ported our mobile-thread based parallel simulation system [14] on the cluster testbed. This system was built on top of a thread library which has thread migration capability. We conducted experiments by running this simulation system to simulate a closed queueing network on the grunt cluster. An $M \times N$ closed queueing network consists of M tandem queues, each containing N FIFO servers. A job which arrives at a queue is served by the N servers sequentially. After completing service at the last server in a queue, the job is routed back to the front of any queue based on a given probability.

In the simulation exercise, we executed a 8×64 closed queueing network model and a 16×256 closed queueing network model, each initialized with a total of 2048 jobs. For simplicity, the probabilities on the arcs routing customers back to the start of each N-server tandem queue are made equal. Table 4.1 shows the performance figures obtained on the grunt cluster, giving both the time in seconds as well as speedup. Particularly good speedup was obtained for the 16×256 network model because of its larger computation granularity.

4.2 The NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) [1] were devised by the Numerical Aerodynamics Simulation (NAS) program at NASA Ames Research Center to study the performance of parallel supercomputers. The NPB 2.3 are a set of eight problems which consists of five kernels which highlight specific areas of machine performance, and three pseudo-applications which simulate computational fluid dynamics(CFD). We briefly describe the benchmarks which we used in our experiments below.

- **Kernel CG** uses the conjugate gradient method to approximate the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. The communication patterns in this benchmark are unstructured and long distance.

- **Kernel EP** generates pairs of Gaussian random deviates and tabulates the number of pairs in successive square annuli. This application is "embarrassingly parallel" because the only communication is summing up a 10-integer list at the end of the program.

- **Kernel IS** performs ranking an unsorted sequence of integer keys that are uniformly distributed among processors. This benchmark requires frequent communication and the pattern of communication is a fully connected graph.

- **Kernel MG** executes four iterations of the V-cycle multigrid algorithm to obtain an approximate solution to the discrete Poisson problem $\nabla^2 u = v$ on a 3-dimensional grid with periodic boundary conditions. The communication patterns are highly structured and both short and long distance data communication are required.

- **Application LU** solves a finite difference discretization of the 3-D compressible Navier-Stokes equations by using a symmetric successive over-relaxation (SSOR) numerical scheme.

- **Application BT** and **SP** are both based on Beam-Warming approximate factorization which decouples the x , y, and z dimensions, resulting in three sets of narrow-banded, regularly structured systems of linear equations. These systems are scalar pentadiagonal in SP, and block tridiagonal in BP.

The NPB 2.3 codes are implemented on top of the MPI library. Table 3 shows the execution time of running seven of the NAS Parallel Benchmarks (class A) using different numbers of workstations. For the purpose of comparison, we also ran the same benchmarks on a 200MHZ Pentium Pro machine (ALR). As can be seen in Table 3, the benchmark performance using one 400 MHZ Pentium II can be two times faster than a 200MHZ Pentium Pro.

There is no doubt that faster microprocessor can improve the performance. Same reason can be applied to a more efficient system software. With the availability of the Portland Group, Inc. Fortran and C compilers, we recompiled the codes and ran the benchmarks. For the purpose of fair comparison, we used the same code optimization level `-O3` which is provided by both of the GNU compilers and the Portland Group software. It can be seen in Table 3 that for certain computational intensive application (like LU), the performance of the codes compiled by the Portland Group compilers can be improved more than 20% than using widely-used GNU compilers.

We also compared the performance of NPB (class B with larger problem size) between different clusters (see Table 4). For communication intensive applications (e.g. IS), the page cluster can run faster than the grunt cluster because it uses faster Gigabit Ethernet. However, the page cluster runs slower for the CG

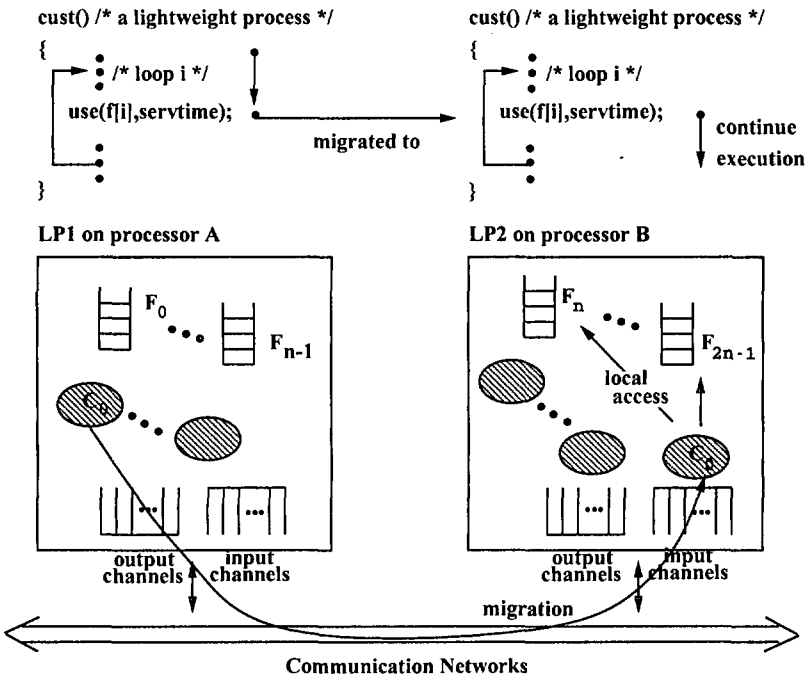


Figure 7: Migrating a thread to the remote machine

Benchmark	1 proc			2 proc		4 proc		8/9* proc		16 proc	
	ALR	gnu	pg	gnu	pg	gnu	pg	gnu	pg	gnu	pg
Block Tridiagonal*	9081	4630	4190			1273	1163	778	729	421	402
Conjugate Gradient	79	69	68	36	36	26	26	24	24	22	22
Embarrassingly Parallel	691	441	437	221	219	103	102	52	52	26	26
Integer Sort	26	24	21	26	25	23	22	18	17	16	16
MultiGrid	196	135	123	71.0	63	40	37	20	17	11	10
Scalar Pentadiagonal*	4166	3233	3032			913	867	508	488	293	281
LU solver	7734	3659	2823	1884	1456	922	766	418	382	240	225

Table 3: Performance of NAS Parallel Benchmarks (class A) on Grunt Cluster (Unit: seconds, *: requires square number of processors)

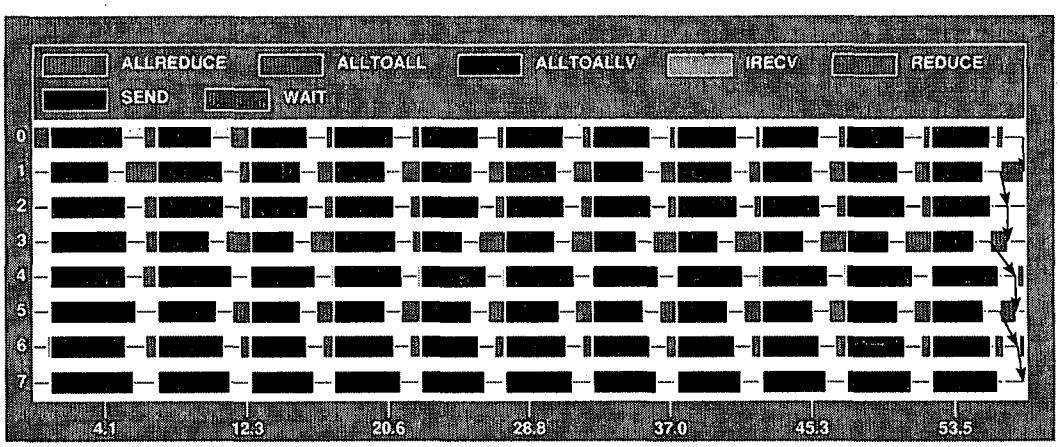


Figure 8: Profile of the IS Benchmark

Benchmark	CG	EP	IS	MG
Grunt (512MB, 100Mbps))	512	196	76	86
Page (128MB, 1000Mbps)	527	195	55	90

Table 4: Performance of NAS Benchmarks (class B) using 8 processors on different clusters (Unit: second)

and MG applications. This may be due to the smaller size of memory on each page machine.

For understanding parallel program behavior, the MPICH library supports profiling functions which are very useful in debugging and in performance analysis. For example, in order to know more about the program behavior of the IS benchmark, we used the option `-mpilog` to compile the code and then invoked the Tcl/Tk script `upshot` to graphically display the timeline of each process. Figure 8 depicts a screen dump for the IS benchmark profile.

It is worth to mention that the users cannot be totally unaware of the cluster structure, especially for running communication intensive applications. Use the IS benchmark as an example. If we selected following eight grunt machines "grunt01, grunt05, grunt09, grunt13, $\cdots$, grunt29 " (i.e. one from each page domain), the performance was very poor because the communication latency was large. More interesting is that, even if we chose the eight machines closely together and specified the following order "grunt01, grunt02, grunt03, grunt04, $\cdots$, grunt08 " in the file `machines.LINUX`, the program execution time was still around 87 seconds. Finally, after we rearranged the order to be "grunt01, grunt05, grunt02, grunt06, $\cdots$, grunt08 ", the performance was improved as shown in Table 4. The reason is due to the better matching between the cluster's tree structure and the "power-of-two" communication pattern in MPICH. Using the profiling function, we found that the communication time was greatly reduced.

5 Conclusion

In this paper, we have extensively measured the performance of cluster computing over Gigabit/Fast Ethernet LANs. Our measurements demonstrate that the high-end 400MHZ Pentium II can easily saturate a 100Mbps Fast Ethernet LAN, but can only generate 20% of the traffic of a Gigabit Ethernet. Since a Gigabit Ethernet can provide enough backbone bandwidth, mixing Gigabit Ethernet and Fast Ethernet for high-performance cluster computing is truly promising.

The flexible tree-structure setting gives us two experimental platforms: a gigabit cluster and a 100Mbps cluster. We have conducted several experiments by running distributed applications on the two platforms. Our measurements show that application performance can be improved by using faster processors, more memory, greater bandwidth networks, or even through better compiler with efficient code optimization capability. Our experiences also suggest that, for communication intensive applications, matching the collective communication patterns with the cluster interconnection network topology can reduce communication overheads and hence improve performance.

Acknowledgements

The authors would like to thank Yan Hu for her help with the network communication measurements. This research was supported by NASA LeRC Summer Faculty Research Fellowship, Ohio Board of Regents equipment grant, Packard Bell Equipment Sponsorship, and CSU EFFRD award.

References

[1] D. Bailey, T. Harris, W. Saphir, and R. Wijngaart. The NAS Parallel Benchmarks 2.0. Technical report, Rech. report NAS-95-020, NASA Ames Research Cneter, 1995.

[2] L. Cabrera, E. Hunter, M. J. Karels, and D. A. Mosher. User-Process Communication Performance in Networks of Computers. *IEEE Trans. on Software Engineering*, 14(1):38–53, Jan. 1988.

[3] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, May 1979.

[4] R. Fujimoto. Parallel discrete event simulation. *CACM*, 33(10):30–53, 1990.

[5] J. C. Gomez, V. Rego, and V. S. Sunderam. Efficient MultiThreaded User-Space Transport for Network Computing: Design and Test for the TRAP Protocol. *Journal of Parallel and Distributed Computing*, 40(1), Jan. 1997.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message passing Interface standard. In *http://www.mcs.anl.gov/mpi/mpicharticle/paper.html, Argonne National laboratory*, 1996.

[7] H. W. Johnson. *Fast Ethernet: Dawn of a New Network.* Prentice Hall, Upper Saddle River, NJ 07458, 1996.

[8] M. Lin, J. Hsieh, D. Du, J. Thomas, and J. MacDonald. Distributed Network Computing over Local ATM Networks. *IEEE Trans. on Selected*

Areas in Communications, 13(4):733–748, May 1995.

[9] A. Matrone, P. Schiano, and V. Puoti. LINDA and PVM: a Comparison between two Environments for Parallel Programming. *Parallel Programming*, 19:949–957, 1993.

[10] Packet Engines. *FDR12 User Guide*, 1997.

[11] D. K. Panda and L. M. Ni. Special Issue on Workstation Clusters and Network-Based Computing: Guest Editors' Introduction. *Journal of Parallel and Distributed Computing*, 40(1), Jan. 1997.

[12] J. Sang, Y. Hu, and M. Tichinel. Experimental Evaluation of Network Computing over Fast Ethernet/ATM LANs. In *Proceedings of the the International Conference on Networks and Communication Systems*, May 1998.

[13] J. Sang, C. Lin, and M. Wang. Experiences with Network-based Computing over Wireless Links. *Int'l Journal of Parallel and Distributed Systems and Networks*, 1999.

[14] J. Sang, E. Mascarenhas, and V. Rego. Mobile-Process Based Parallel Simulation. *Journal of Parallel and Distributed Computing*, February 1996.

[15] R. Seifert. *Gigabit Ethernet.* Addison-Wesley, Reading, Mass., 1 edition, 1998.

The Remote Enqueue Operation on Networks of Workstations

Manolis G.H. Katevenis[1], Evangelos P. Markatos[1], Penny Vatsolaki and Chara Xanthaki
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH), Crete
P.O.Box 1385 Heraklio, Crete, GR-711-10 GREECE
http://archvlsi.ics.forth.gr

Keywords: cluster computing, networks of workstations, high-performance computing, message passing

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 5, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

Modern networks of workstations connected by Gigabit networks have the ability to run high-performance computing applications at a reasonable performance, but at a significantly lower cost. The performance of these applications is usually dominated by their efficiency of the underlying communication mechanisms. However, efficient communication requires that not only messages themselves are sent fast, but also notification about message arrival should be fast as well. For example, a message that has arrived at its destination is worthless until the recipient is alerted to the message arrival.

In this paper we describe a new operation, the remote-enqueue *atomic operation, which can be used in multiprocessors, and workstation clusters. This operation atomically inserts a data element in a queue that physically resides in a remote processor's memory. This operation can be used for fast notification of message arrival, and for fast passing of small messages. Compared to other software and hardware queueing alternatives, remote-enqueue provides high speed at a low implementation cost without compromising protection in a general-purpose computing environment.*

1 Introduction

Popular contemporary computing environments are comprised of powerful workstations connected via a network which, in many cases, may have a high throughput, giving rise to systems called *workstation clusters* or Networks of Workstations (NOWs) [1]. The availability of such computing and communication power gives rise to new applications like multimedia, high performance scientific computing, real-time applications, engineering design and simulation, and so on. Up to recently, only high performance parallel processors and supercomputers were able to satisfy the computing requirements of these applications. Fortunately, modern workstations connected by Gigabit networks have the ability to run most applications that run on supercomputers, at a reasonable performance, but at a significantly lower cost. This is because most modern Gigabit interconnection networks provide both low latency and high throughput. However, efficient communication requires that not only messages themselves are sent fast, but also notification about message arrival should be fast as well. For example, a message that has arrived at its destination is worthless until the recipient is alerted to the message arrival.

In this paper we present the *Remote Enqueue* atomic operation, which allows user-level processes to enqueue (short) data in remote queues that reside in various workstations in a cluster, with no need for prior synchronization. This operation was developed within the Telegraphos project [18], in order to provide a fast message arrival notification mechanism. The Telegraphos network interface provides user applications with the ability to read/write remote memory locations, using regular `load/store` instructions to remote memory addresses. Sending (short) messages in Telegraphos can be done by issuing one or more remote write operation, which eliminates traditional operating system overheads that used to dominate message passing. Thus, sending (short) messages can be done from user-level by issuing a few `store` assembly instructions. Although sending a message can be done fast, notifying the recipient of the message arrival may take significant overhead. For example, one might use a shared flag in which the sender writes the memory location (in the recipient's memory) where (and when) the message was written. When the recipient checks for messages, it reads this shared flag and finds out if there is an arrived message and where it is. However, if two or more senders attempt to send a message at about the same time, only one of them will manage to update the flag, and the other's update will be lost. A solution would be to have a separate flag for

[1]Evangelos P. Markatos and Manolis G.H. Katevenis are also with the University of Crete.

each possible sender. However, if there are several potential senders, this solution may result in significant overhead for the receiver, who would be required to poll too many flags. Arranging the flags in hierarchical (scalable) data structures might reduce the polling overhead, but it would increase the message notification arrival overhead.

Our solution to the message arrival notification problem is to create a remote queue of message arrival notifications. A remote queue is a data structure that resides in the remote node's main memory. After writing their message to the receiver's main memory, senders enqueue their message arrival notifications in the remote queue. Receivers poll their notification queues to learn about arrived messages. Although enqueueing notifications in remote queues can be done completely in software, we propose a hardware *remote enqueue* operation that *atomically* enqueues a message notification in a remote queue. The benefits of our approach are:

- *Atomicity at low cost*: to prevent race conditions, all software-implemented enqueue operations are based on locking (or on `fetch_and_`ϕ atomic) operations that appropriately serialize concurrent accesses to the queue. These operations incur the overhead of at least one network round-trip delay. Our hardware-implemented remote enqueue operation serializes concurrent enqueue operations at the receiver's network interface, alleviating the need for round-trip messages.

- *Low-latency flow control:* Most software enqueue operations may delay (block) the enqueing process if the queue is full. For this reason, most software-implemented enqueue operations need to read some metadata associated with the remote queue in order to make sure that the remote queue is not full. Unfortunately, reading remote data may take at least one round-trip network delay. In our approach, the enqueueing operation *always* succeeds; if the queue fills up after an enqueue operation, a software handler is invoked (at the remote node) to allocate more space for the queue. Since our remote enqueue operation is non-blocking, and does not need to read remote data, it can return control to its calling processes, as soon as the data to be enqueued have been entered in the sender's network interface, that is the remote enqueue operation may return control within a few (network interface) clock cycles - usually a fraction of a microsecond.

The rest of the paper is organized as follows: Section 2 surveys previous work. Section 3 presents a summary of the Telegraphos workstation cluster. Section 4 presents the remote enqueue operation, and section 5 summarizes this paper.

2 Related Work

Although networks of workstations may have an (aggregate) computing power comparable to that of supercomputers (while costing significantly less), they have rarely been used to support high-performance computing, because communication on them has traditionally been very expensive. There have been several projects to provide efficient communication primitives in networks of workstations via a combination of hardware and software: Dolphin's SCI interface [19], PRAM [24], Memory Channel [13], Myrinet [6], ServerNet [26], Active Messages [12], Fast Messages [17], Galactica Net [16], Hamlyn [9], U-Net [27], PCI-DDC [28], NOW [1], Parastation [29], StarT Jt [15], Avalanche [10], Panda [2], SHRIMP [4] and others provide efficient message passing on networks of workstations usually based on memory-mapped interfaces. We view our work as complimentary to these projects. Most of them have developed novel efficient mechanisms to send data between two different workstations in a cluster. We complement the mentioned previous work by proposing a fast message notification mechanism that can improve the performance of all these message passing systems.

Brewer *et. al* proposed *Remote Queues*, a communication model that is based on enqueueing and dequeuing information in queues in remote processors [8]. Their model is mostly software based, but it can also it can be tuned to exploit any existing hardware mechanisms (e.g. hardware queues) that may exist in a parallel machine. Although their work is related to ours we see two major differences:

- *Remote queues* combine message transfer with message notification: the message itself is enqueued in the remote queue. The receiver reads the message from the queue and (if appropriate) copies the message to its final destination in its local memory. In our approach we assume that the message has been posted directly to its final destination in the receiver's memory, and only the notification of the message arrival need to be put in the queue - our approach results in less message copy operations. Suppose for example that the sender and the receiver share a common data structure (e.g. a graph). Using out approach, the sender deposits its information directly in the remote graph, where the receiver will read it from. On the contrary, in the remote queues approach, the messages are first placed in a queue, and the receiver will have to copy the messages from the queue and put their information on the common graph, resulting in one extra copy operation. Recent commercial network interfaces like the PCI-DDC [28], the Memory Channel [13] and the PCI-SCI [19] efficiently support our approach of the direct deposit of data in the receiver's memory.

– *Remote Queues* have been designed and implemented in commercial and experimental massively parallel processors that run parallel applications in a controlled environment, supporting little or no multiprogramming. Our approach has been designed for low-cost Networks of Workstations that support both sequential and parallel applications that need to be separated (protected) from each other. Designing for a general-purpose multiprogrammed environment is considerably more difficult and complicates several details of the design.

In single-address-space multiprocessors, our remote enqueue operation can be completely implemented in software using any standard queue library. Brewer *et. al* propose such an implementation on top of the Cray T3D shared-memory multiprocessor [8]. Any such implementation (including the one in [8]) suffers from software overhead that includes at least one atomic operation (to atomically get an empty slot in the queue), plus several remote memory accesses (to place the data in the remote queue and update the remote pointers). This overhead is bound to be significant in a Network of Workstations.

In many multiprocessors, nodes have a network co-processor. In these cases, the remote enqueue operation can be implemented with the help of this co-processor. The co-processor implements sophisticated forms of communication with the processes running on the host processor. For example, a process that wants to enqueue a message in a remote queue, sends the message to the co-processor, which forwards it to the co-processor in the remote node, which in turn places the message in the remote queue. Although the existence co-processors improves the communication abilities of a node, it may result (i) in software overhead (after all they are regular microprocessors executing a software protocol), (ii) in more data copy operations, and (iii) in increased end-system cost.

3 The Telegraphos NOW

The Remote enqueue operation described in this paper is developed within the Telegraphos project [22]. *Telegraphos* is a distributed system that consists of network interfaces and switches for efficient support of parallel and distributed applications on a workstation cluster (shown in Figure 1). We call this project Telegraphos or $T\eta\lambda\acute{\epsilon}\gamma\rho\alpha\phi o\varsigma$ from the greek words $T\eta\lambda\acute{\epsilon}$ meaning remote, and $\gamma\rho\acute{\alpha}\phi\omega$ meaning write, because the *central* operation on Telegraphos is the remote write operation. A remote write operation is triggered by a simple `store` assembly instruction, whose argument is a (virtual) memory address mapped on the physical memory of another workstation. Figure 2 shows illustrates how process P_i sends a message to processor

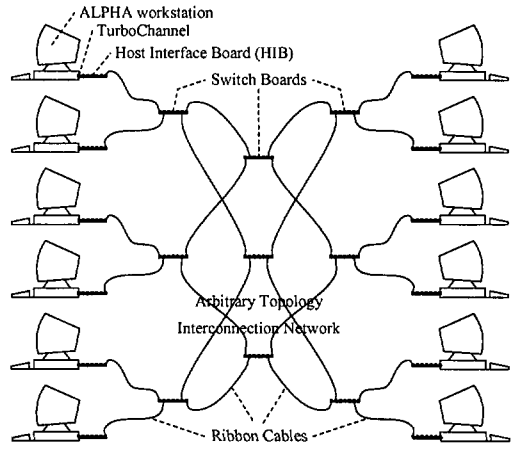


Figure 1: **The Telegraphos Workstation Cluster.**

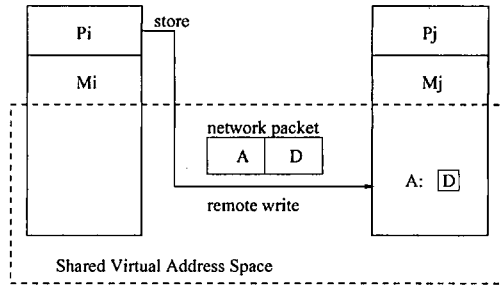


Figure 2: **Message Passing by Remote Writes.** Processor P_i issues a `store` instruction, which is snooped by the network interface. The address A and the data D of the instruction are packed in a message and sent to processor P_j in a network packet. When the packet reaches its destination, processor P_j will store the data D into its address A.

P_j by issuing an assembly `store` instruction. The remote write operation makes possible the (user-to-user, fully protected) sending of short messages with a *single* instruction. For comparison, traditional workstation clusters connected via FDDI and ATM take several thousands of instructions to send even the shortest message across the network.

Telegraphos provides a variety of hardware primitives which, when combined with appropriate software will result in efficient support for shared-memory applications. These primitives include:

– *Single remote memory access*: On a *remote* memory access, traditional systems require the help of the operating system, which either replicates locally the remote page and makes a local memory access, or makes the single remote access on behalf of the requesting process. To avoid this operating system overhead, Telegraphos provides the processor with the ability to make a read or write operation to a remote memory location without replicating the page locally and without any software intervention; just like shared-memory multi-

processors do [3].

- *Access counters:* If a page is accessed by a processor frequently, it may be worthwhile to replicate the page and make all accesses to it locally. To allow informed decisions, Telegraphos provides access counters for each remotely-mapped page. Each time the processor accesses a remote page, the counter is decremented, and when it reaches zero an interrupt is sent to the processor which should probably replicate the page locally [7, 20, 21].

- *Hardware multicasting:* Telegraphos provides a write multicast mechanism in hardware which can be used to implement one-to-many message passing operations, as well as an update-based memory coherence protocol. This multicast mechanism uses a novel memory coherency protocol that makes sure that even when several processors try to update the same data and multicast their updates at the same time, they will all see a consistent view of the updated data; details about the protocol can be found at [22].

- *User-level DMA:* To facilitate efficient message passing, Telegraphos allows user-level initiation of all shared-memory operations including DMA. Thus, Telegraphos does not need the involvement of the Operating System to transfer information from one workstation to another [23].

The Telegraphos network interface has been prototyped using FPGA's; it plugs into the TurboChannel I/O bus of DEC Alpha 3000 model 300 (Pelican) workstations.

4 Remote Enqueue

4.1 Message Passing via Remote Writes

In a shared-address space multiprocessor that supports efficient remote-write operations (like Telegraphos) message passing is performed by issuing remote-write operations from user space. In this way, the sender directly deposits the message to its final destination in remote memory, which is accessed using regular load and store assembly instruction (just like local memory). Although message passing via remote-writes is very efficient, notifying the sender that a message has arrived becomes complicated. Figure 3 shows several potential senders P_1 to P_N and one receiver P. Each sender has mapped a buffer in P's local memory and sends messages to P by writing the message directly to this buffer. However, it becomes increasingly difficult for P to know when and from which source a message has arrived. One could propose that P gets

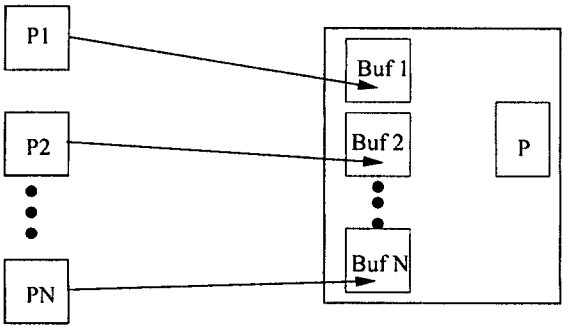


Figure 3: **Multi-source message arrival notification.**

interrupted each time a message arrives. However, interrupts would add significant overhead to message-passing, since they are usually handled by the operating system kernel, which would undermine the efficiency of user-level message passing. To avoid the operating system involvement, we could use one "arrival bit" for each buffer. When a message is sent, the corresponding arrival bit is set as well. However, when the number of potential senders becomes high, polling a large number of arrival bits would jeopardize the scalability of this solution. To achieve a scalable notification mechanism we could use a tree of arrival bits, but such a mechanism would complicate programming and increase the latency of message arrival notification. What we propose to do is to have a queue of message arrival notifications. For each new message, the sender enqueues a notification in the queue. When the local processor wants to check for messages, it reads the head of the notification queue. Although the queue can be managed completely in software, we propose that enqueueing should be done in hardware using the "remote enqueue" operation, while dequeuing can be done in (user-level) software. We avoid a completely-software solution since this would require synchronization using atomic operation which would add significantly to the overhead of enqueueing.

4.2 The Remote Enqueue Operation

We propose a new atomic operation, the *remote enqueue (REQ)* atomic operation. The REQ atomic operation is invoked with two arguments:,

- *REQ(vaddress,data)*, where *vaddress* is the virtual address that uniquely identifies a remote queue (a remote queue always resides on the physical memory of a different processor from the one invoking the REQ operation), and *data* is a single word of information to be inserted in the queue. This information is most usually a virtual address (pointer) that identifies the message body that the processor invoking the REQ operation has just sent to the processor that hosts the queue in its

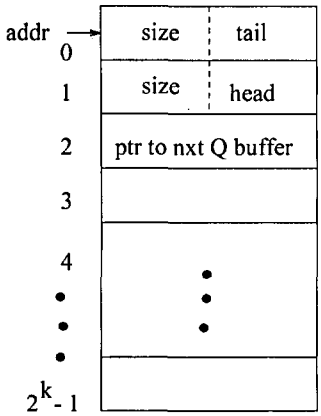


Figure 4: **Layout of a data buffer.** A remote queue is just a linked list of such buffers. The first three words of the buffer are reserved to store the size, the tail, the head, and the pointer to the next Q buffer.

memory. [1]

We define a *remote queue* to be a portion of a remote processor's memory that is managed as a FIFO queue. This FIFO queue is a linked list of buffers which are physically allocated in the remote processor's memory. Data are placed in this FIFO queue by the *remote enqueue (REQ)* operation, implemented in hardware. Data are removed from this FIFO queue with a *dequeue* operation which is implemented in user-level software.

The following limitations are imposed to the buffers of a remote queue, for the hardware remote enqueue operation to be efficient:

- The starting address of each buffer should be an integer multiple of the buffer size, which is a power of two.
- The maximum buffer size is 64KB (for a 32-bit word processor).
- If the buffer size is larger than the page size, each buffer should be allocated in contiguous physical pages.

The layout of the buffer is shown in figure 4. The *head* and *tail* are indices in the *data buffer*. A queue is a linked list of such data buffers. For a 32-bit-word processor, both *tail*, and *head* are 16 bit quantities, and not full memory addresses. The reason is that, in traditional systems (where *tail* and *head* are full addresses), we calculate the pointer to the head (or the tail) of the queue by adding *addr* with *head* (or *tail*), meaning that we need to pay the hardware cost of an extra adder, and the performance cost of a word-length addition. In our system instead, where the *addr* is a multiple of a power of two, and both *head* and *tail* are always less than this power of two, we calculate the pointer to the head (or the tail) of the queue by performing an inexpensive *OR* operation instead of an expensive addition.

[1]The Telegraphos network always delivers remote data *in-order* from a given source to a given destination node. Thus, data can never arrive before the corresponding REQ operation is posted.

4.3 The Enqueue Operation

When processor A wants to enqueue some *data* in the remote queue that starts in the virtual address *vaddress* and that that physically resides on processor B's memory, it invokes the *REQ(vaddress,data)* atomic operation. A portion of this operation in implemented on the sender node's network interface, and the rest is implemented on the receiver node's network interface.

The Sender Node: When the software issues a REQ(vaddress,data) atomic operation, the local network interface takes the following actions:

- It prepares a *remote-enqueue-request* packet to be sent to the remote node that contains *paddress* (the physical address that corresponds to virtual address *vaddress*), and *data*, and
- It releases the issuing processor, which is able to continue with the rest of its program, without having to wait for the remote enqueue operation to complete.

The Receiver Node: When the destination node receives a *remote-enqueue-request* packet it extracts the *paddress* and *data* arguments from the packet and performs the remote-enqueue operation as the following atomic sequence of steps:

- Writes the *data* to the buffer entry pointed by the *tail* index (the address of the entry is calculated as *(paddress OR tail)*.
- Increments the *tail* by 1 modulo buffer *size* (if the *tail* equals the *size* of the buffer, then *tail* gets the value of the first available buffer location: 3 (see figure 4)).
- If the buffer overflows (*tail* = *head*), the network interface stops accepting incoming network requests, and sends an interrupt to the (destination) processor.

The hardware finite state machine (FSM) of the destination HIB for the remote enqueue operation "req(addr, data)" is shown in table 1.

```
     FSM()
1.   read  (address)        -> (size, tail) // read tail and size of Q
2.   write (address OR tail)<- (data)  // insert new element in Q
     // Note: (address OR tail) points to the first free element in the Q
     // thus: no adder is needed
3.   tmp                 <- (tail + 1) // increment tail modulo size
     // if (tmp == size) then tail = 3
4.   if (tmp & size)     then
           tail <- 3
     else
           tail <- tmp
     // Note: if (tmp == size) then (tmp & size) == 1
     // else  (tmp & size) == 0,
     // thus the comparison can be implemented with AND
     // gates instead of a general purpose comparator
5.   read  (address+1)      -> (size,head) // read head of Q
6.   if (head == tail) then
           stop_accepting_network_requests()
           interrupt host (overflow)
     else
           write (address)  <- (size,tail)
```

Table 1: Finite State Machine for the Remote Enqueue Operation.

Hardware Diagram: The Telegraphos datapath for the remote enqueue operation (at the receiver side) is shown in figure 5. The whole operation is controlled by five control signals: LD0, RD0, WR0, RD1, and WR1, that are generated by a simple Finite State Machine in the above order.

- LD0 loads the ADDRESS and DATA registers with the address and data that are the arguments of the remote enqueue operation.
- RD0 starts the reading of the $(size, tail)$ pair from *address*.
- WR0 starts the writing of the *data* into the remote queue at address *address OR tail*
- RD1 starts the reading of the $(size, head)$ pair from $(address + 1)$.
- Finally, WR1 writes the new $(size, tail)$ pair into *address*

4.4 Handling Buffer Overflow

When the current buffer fills up, an interrupt is sent to the processor which starts executing the operating system. The actions that the operating system should take are:

- Copy the contents of the full buffer into an empty one. Mark the previously overflowed buffer as empty.
- Link the new buffer into a queue of buffers associated with this queue. The *next* field in the queue is used for this purpose.
- Enable the Network Interface to handle all requests.

4.5 Dequeuing and Queue Handling in the Receiver Software

In this section we outline how the dequeue operation can be efficiently implemented in software at user-level. A straightforward implementation of the dequeue operation would be:

```
deq(queue)
{
    buffer = find_last_buffer() ;
    if (is_empty(buffer) {
       if (is_first(buffer, queue))
           return EMPTY_QUEUE ;
       else  {
           deallocate(buffer) ;
           buffer = find_last_buffer() ;
       }
    }
    result =  buffer[head] ;  head ++ ;
    if (head == size)
        head = 3 ;
    return result
}
```

Unfortunately, the above solution does not always work, because it is executed in user-space, and as such,

it may be interrupted at any time. For example, consider the following scenario:

- A dequeue operation starts executing, taking an element from the head buffer (say A) of the queue.
- Before the operation completes, it is interrupted.
- In the meanwhile, the head buffer overflows, the operating system takes control, copies the buffer A into an empty one (say B), resetting the previously full buffer A.
- Some more remote enqueue operations are executed, completely overwriting the previous data on A (which have been safely copied into the recently allocated buffer B).
- The dequeue operation eventually resumes execution trying to dequeue elements from buffer A, which does not have the elements the dequeue operation expects to find, which are now in buffer B!

Fortunately, on the DEC Alpha processor there is a special mode the *PAL mode* which enables (super) users to write their own code (of limited size) and run it uninterrupted [25]. Thus, if the above code is turned into PAL code, it will run uninterrupted. PAL code is invoked via the special *cal_pal* routine, that the DEC Alpha processor provides. Although any user is allowed to call a PAL function, only the super user is allowed to install new PAL functions, thereby protecting the integrity of the system. Thus, the above mentioned race conditions disappear because the dequeue operation runs uninterrupted in PAL mode.

Although PAL calls are an elegant way of executing short sequences of instructions uninterrupted, they are specific to the Alpha processor. Moreover, interrupt disabling (and of course PAL calls) is an effective way of synchronization only in uniprocessors. Disabling interrupts in symmetric multiprocessing systems that share a common network interface does not necessarily guarantee the absence of race conditions. For this reason, we have developed a more general solution that allows dequeue operations to proceed at user-level without the need to invoke PAL calls. Our solution is based on the collaboration between the operating system and the library that implements the dequeue operation. We assume the existence of a "do-not-preempt-me" bit (per queue) that is shared by the user application and the kernel. [2] When the application is about to execute a dequeue operation, it sets the "do-not-preempt-me" bit. When the dequeue operation completes, it resets the "do-not-preempt-me" bit. If the queue becomes full while an application is dequeuing something from the queue, the operating system driver that handles the buffer overflow interrupt, does not allocate a new buffer but sets a "full-queue" flag. When the interrupt handler returns, the application will resume execution, and it will complete the dequeue operation. When the dequeue operation completes, it checks the "full-queue" flag. If the flag is set, the application will invoke the network interface driver (e.g. through an `ioctl` call) to allocate more space for the queue and to enable the network interface to handle further enqueue operations. This solution works even in multiprocessor workstations that share a single network interface, with only one additional requirement: threads that execute concurrent dequeue operations (from the same queue) have to synchronize through a lock variable (associated with the queue). The first instruction of a dequeue operation is to acquire the lock, and the last instruction is to release the lock. Thus, while a thread is dequeuing data from a queue, no other thread is allowed to do the same, and thus no other thread can access shared information like the "do-not-preempt-me" bit and the "full-queue" flag. In case of buffer overflow, user-level threads should keep the lock up till the time the operating system allocates more space for the queue. If the queue fills up while at the same time a thread is executing a dequeue operation, the operating system allows the dequeue operation to complete; after the operation completes it invokes the operating system to allocate more space for the queue and to enable further network transactions.

[2] Similar mechanisms has been used to avoid preempting a user-level thread while executing in a critical section [11].

4.6 Issuing an Enqueue Operation

An enqueue operation is invoked as:
enq(vaddress, data)
(where *vaddress* is the virtual address of the base of the first queue buffer and *data* are the data to be enqueued). In order to create a valid *remote enqueue request* packet, the network interface needs to know the *physical* address *paddress* that corresponds to virtual address *vaddress*, as well as the *data* argument. However, users are not allowed to communicate physical addresses to the network interface, because (i) they no dot know the mapping between virtual and physical pages, and (ii) malicious or ignorant users may request enqueue operations to physical addresses on which they do not have read/write access. To alleviate this problem we use the mechanism of *shadow-addressing* [5, 14, 23]. The method of shadow addressing is used to securely translate virtual to physical addresses and pass them to the network interface from user-level processes. For each virtual address `vaddress` that is mapped in the physical address `paddress`, there is also a shadow address `shadow(vaddress)`, which is mapped in the shadow

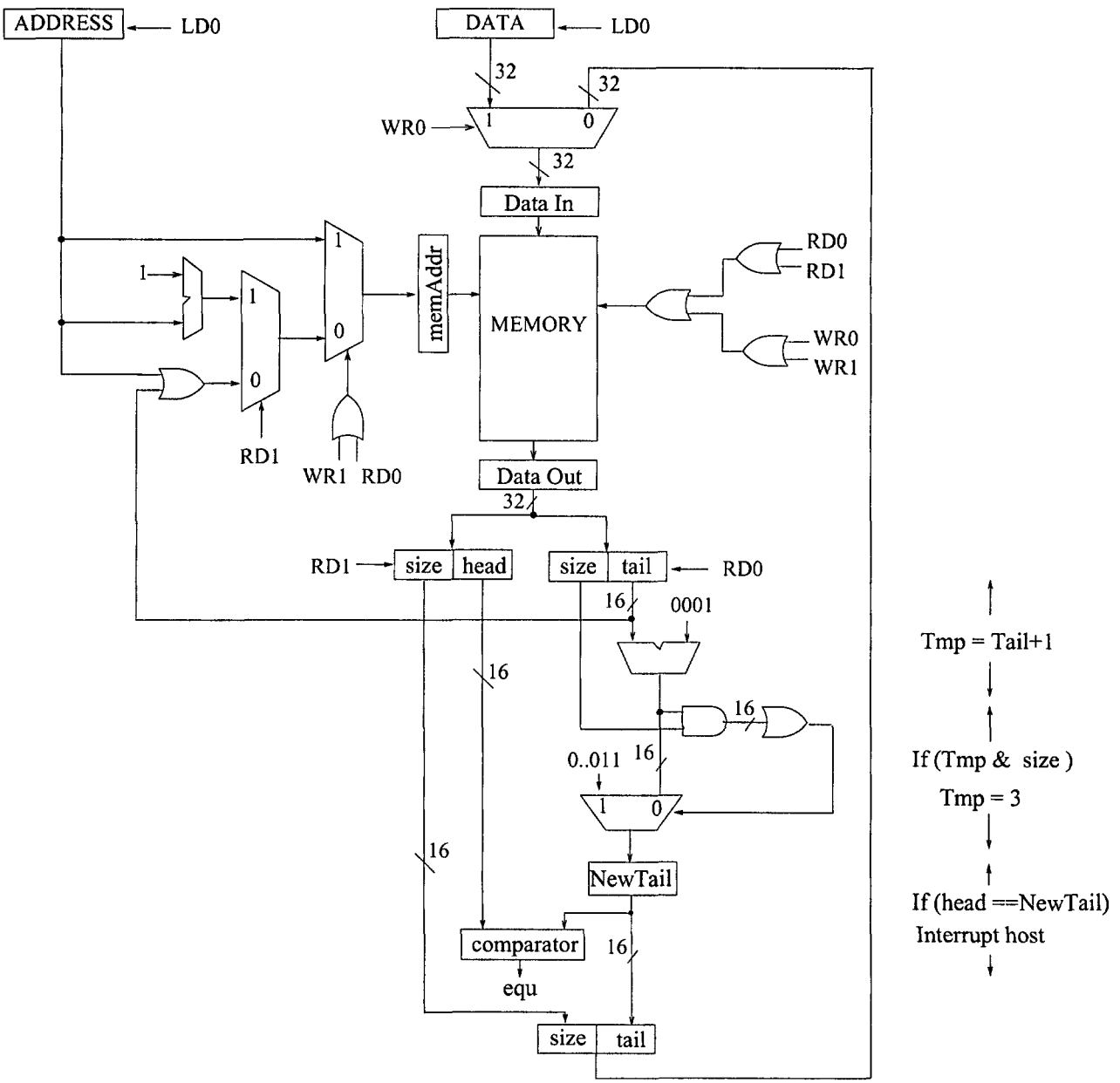


Figure 5: **The enqueue hardware.**

physical address `shadow(paddress)`.[3] The shadow function is simple and known to the network interface. One simple shadow function is to concatenate each address with an extra shadow bit. When the shadow bit is set, then the address is a shadow one. For example, 0x0FFFFFFFF is a regular 33-bit address, while 0x1FFFFFFFF is its shadow address.

An access to a shadow address is always interpreted by the network interface as a special argument passing operation. For example, suppose that virtual address `vaddress` is mapped to physical address `paddress`, and that the virtual address `shadow(vaddress)` is mapped into `shadow(paddress)`. Normally, a load (store) operation to virtual address `vaddress` by a user application is translated by the TLB (page-table) into a load (store) operation to physical address `paddress` and is performed by the appropriate memory controller. Similarly, a load (store) operation to virtual address `shadow(vaddress)` is translated by the TLB into a load (store) operation to physical address `shadow(paddress)`. When, however, this operation reaches the network interface it will be treated as an argument passing operation, and neither a load nor a store operation will be performed to physical address `shadow(paddress)`. Thus, when the user application wants to pass to the network interface the physical address `paddress`, it makes a store operation to virtual address `shadow(vaddress)`. After TLB translation the physical address `shadow(paddress)` reaches the network interface, which recognizes the shadow address and takes the physical address `paddress` by applying function $\texttt{shadow}^{-1}$ to physical address `shadow(paddress)`.[4]

Thus, a remote enqueue atomic operation is issued using a single assembly instruction as follows:

```
REQ (vaddress, data)
    /* pass physical address shadow(paddress)
    ** to the network interface */
    STORE data TO shadow(vaddress)
```

The processor latency of a remote write operation on Telegraphos has been measured to be only 0.7 microseconds. Thus, the message-notification overhead observed by the sending processor is very small - comparable to the latency of a local memory access.

5 Summary

In this paper we describe a new operation, the *remote-enqueue* atomic operation, which can be used in multiprocessors, and workstation clusters. This operation atomically inserts a data element in a queue that physically resides in a remote processor's memory. This operation can be used for fast notification of message arrival, and for fast passing of small messages. Both enqueue and dequeue operations can be issued from user-level processes without any need to call the operating system. Both operations enforce standard virtual memory protection when accessing remote queues, and thus they provide full protection in a general-purposed multiprogrammed environment. Compared to other software and hardware queueing alternatives, remote-enqueue provides high speed at a low implementation cost without compromising protection in a general-purpose computing environment.

[3]The Operating System is responsible for creating both mappings at memory allocation (initialization) time.

[4]All shadow addresses should be within the physical address range of the network interface, and distinct from the normal physical addresses used by that network interface.

Acknowledgments

This work was supported in part by ESPRIT project 6253 "Supercomputer Highly Parallel System" (SHIPS), funded by the European Union, through DG III of its Commission, HPCN Unit. We deeply appreciate this financial support, without which this work would have not existed. A patent application for the above work has been filed: E. Markatos, M Katevenis, and P. Vatsolaki: "Notification of message arrival in a parallel computer system", Patent application number 97410036.4, (Europe) March 19th 1997.

Telegraphos is a collective effort, with many contributors. The authors wish to acknowledge in particular Apostolos Dollas, George Kalokairinos, Manolis Stratakis, Chara Xanthaki, and George Papadourakis, for the design and implementation of Telegraphos I and II. Richard Fortier, James Goodman, Robert Hyerle, Alasdair Rawsthorne, and Marios Mavronicolas have helped in various ways, at various times. Kosmas Papachristos and George Dramitinos implemented low-level operating system software for Telegraphos. Finally, the comments of the anonymous reviewers were valuable. We thank all of them.

References

[1] T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] H. Bal, R. Hofman, and K. Verstoep. A Comparison of Three High Speed Networks for Parallel Cluster Computing. In *Proc. 1st International Workshop on Communication and Arch. Support for Network-Based Parallel Computing*, pages 184–197, 1997.

[3] BBN Advanced Computers Inc. *Inside the TC2000^{TM} Computer.* Cambridge, Massachusetts, February 1990.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. 21-th International Symposium on Comp. Arch.*, pages 142–153, Chicago, IL, April 1994.

[5] M.A. Blumrich, C.Dubnicki, E.W. Felten, and K. Li. Protected, User-level DMA for the SHRIMP Network Interface. In *Proc. of the 2nd International Symposium on High Performance Computer Architecture*, pages 154–165, San Jose, CA, February 1996.

[6] N.J. Boden, D. Cohen, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29, February 1995.

[7] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, Santa Clara, CA, April 1991.

[8] E.A. Brewer, F.T. Chong, L.Tl Liu, S.D. Sharma, and J.D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Symp. on Parallel Algorithms and Architecures*, 1995.

[9] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. Hamlyn: a High-performance Network Interface, with Sender-Based Memory Management. In *Proceedings of the Hot Interconnects III Symposium*, August 1995.

[10] A. Davis, M. Swanson, and M. Parker. Efficient Communication Mechanisms for Cluster Based Parallel Computing. Technical report, University of Utah, Dept. of Computer Science, 1996.

[11] J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. Technical Report Ultracomputer Note 136, Ultracomputer Research Laboratory, New York University, April 1988.

[12] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. 19-th International Symposium on Comp. Arch.*, pages 256–266, Gold Coast, Australia, May 1992.

[13] R. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12, February 1996.

[14] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proc. of the 6-th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, 1994.

[15] James C. Hoe and Mike Ehrlich. StarT-JR: A Parallel System from Commodity Technology. In *Proceedings of the 7th Transputer/Occam International Conference*, November 1995. Tokyo, Japan.

[16] Andrew W. Wilson Jr., Richard P. LaRowe Jr., and Marc J. Teller. Hardware Assist for Distributed Shared Memory. In *Proc. 13-th Int. Conf. on Distr. Comp. Syst.*, pages 246–255, Pittsburgh, PA, May 1993.

[17] V. Karamcheti, S. Pakin, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing 95*, 1995.

[18] Manolis G. H. Katevenis, Evangelos P. Markatos, George Kalokerinos, and Apostolos Dollas. Telegraphos: A Substrate for High-Performance Computing on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 43(2):94–108, June 1997.

[19] O. Lysne, S. Gjessing, and K. Lochsen. Running the SCI Protocol over HIC Networks. In *Proceedings of the Second International Workshop on SCI-based Low-cost/High-perfocmance Computing (SCIzzL-2)*, March 1995. Santa Barbara, CA.

[20] E.P. Markatos. Using Remote Memory to avoid Disk Thrashing: A Simulation Study. In *Proceedings of the ACM International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '96)*, pages 69–73, February 1996.

[21] E.P. Markatos and C.E. Chronaki. Trace-Driven Simulations of Data-Alignment and Other Factors affecting Update and Invalidate Based Coherent Memory. In *Proceedings of the ACM International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 44–52, January 1994.

[22] E.P. Markatos and M. G.H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proc. of the 2nd International Symposium on High Performance Computer Architecture*, pages 144–153, Feb 1996. URL: http://www.csi.forth.gr/ proj/arch-vlsi/papers/ 1996.HPCA96.Telegraphos.ps.gz.

[23] E.P. Markatos and M. G.H. Katevenis. User-Level DMA without Operating System Kernel Modification. In *Proc. of the 3rd International Symposium on High Performance Computer Architecture*, pages 322–331, Feb 1997. URL: http://www.csi.forth.gr/proj/aavg/papers/1997.HPCA97.user_level_dma.ps.gz.

[24] D. Serpanos. *Scalable Shared-Memory Interconnections.* PhD thesis, Princeton University, Dept. of Computer Science, October 1990.

[25] R. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[26] Tandem Computers Inc. ServerNet Technology: Introducing the Worlds First System Area Network, 1996. http://www.tandem.com/INFOCTR/BRFS_WPS/SNTSANWP/SNTSANWP.HTM.

[27] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. 15-th Symposium on Operating Systems Principles*, pages 40–53, December 1995.

[28] F. Wajsbort, J-L. Desbarbieux, C. Spasevski, S. Penain, and A. Greiner. An Integrated PCI Component for IEEE 1355. In *European Multimedia, Microprocessor Systems and Electronic Commerce Conference and Exhibition (EMMSEC'97)*, Nov. 1997.

[29] Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. The ParaPC / ParaStation Project: Efficient Parallel Computing by Clustering Workstations. Technical Report 13/96, University of Karlsruhe, Dept. of Informatics, 1996.

Preserving Mutual Interests in High Performance Computing Clusters

Orly Kremien, Kemelmakher Michael and Eshed Irit
Distributed Systems Group
Department of Computer Sciences and Mathematics
Bar Ilan University, Ramat Gan, Israel
Phone : +972-3-5318052 Email : orly,kemelma,eshedi,dsg@macs.biu.ac.il
WWW : http://www.cs.biu.ac.il:8080/dsg

Keywords: adaptability, caching, locality, PVM, pluggability, proximity, resource sharing, scalability

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 15, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

Massive network systems spanning grand geographical distances, like Internet, aim at providing scalable resource access. Requests for resource access in such complex systems randomly arrive at nodes. Cooperation and negotiation are required in order to better support resource sharing, i.e. to decide whether to initiate processing of a resource request locally or locate a remote resource and negotiate for its remote access. The problems encountered in such complex systems are briefly described in this paper. A measurement study in PVM is used to illustrate our approach. PVM uses round-robin as its default policy for process allocation to processors. The main drawbacks of this policy are the fact that PVM ignores load variations among nodes and also the inability of PVM to distinguish between machines of different speeds. To repair this deficiency a Resource Manager (RM) is implemented which replaces round-robin with a scalable and adaptive algorithm for resource sharing providing a High Performance Computing Cluster (HPCC). We propose an implementation of a Resource Manager in PVM. The RM can be transparently plugged into PVM to offer improved performance for its users. The design of a resource manager to extend PVM is outlined. A prototype implementation in PVM is then measured to illustrate the utility of our approach. Performance results favorably comparing our extended RM to the original PVM are presented. In conclusion, our RM is extended to further expedite performance with enhanced locality

1 Introduction

Both computing and networking areas are rapidly changing. High speed networks and improved microprocessor performance are making the network inseparable from the computers it links together. High capacity clusters of workstations are becoming an appealing vehicle for parallel computing. In this environment most machines are autonomous personal workstations where each is dedicated primarily to serving its owner. By interconnecting multiple local clusters through a high-speed communication, very large parallel systems can be built, at low additional cost, creating a large parallel-computing machine.

These complex systems provide access to a variety of resources. There are standard resources as well as nomadic ones. Local access suffices for some whereas others are far off requiring distant access. Furthermore, there are multiple instances greatly differing in characteristics which are accessible for each resource type. A complex network system may be viewed as a collection of services. There are workstations, which seek services and those available, which provide services. Resource sharing in complex network systems aims at achieving maximal system performance by utilizing the available system resources efficiently. The goal is to match workstations short of a service to those experiencing a surplus of the same service. The relationship between such a pair is termed mutual interest. This paradigm is used to propose a scalable and adaptive resource sharing service.

Requests for resource access randomly arrive at nodes. It is possible for some of the nodes equipped with the resource to have it utilized to its fullest capacity executing resource access requests. Other nodes might not be equipped with the resource at all or possibly have its request queue lightly loaded or even idle. This calls for some type of cooperation and negotiation in order to better support resource sharing, i.e. whether to initiate processing of a resource request locally or locate a remote node from a large set of available ones and negotiate for its remote access. Much of the computing power is frequently idle. There is a necessity to coordinate concurrent access to system

resources. Without any mechanism for cooperation among nodes, it is likely that resources on one node will become congested while other nodes are idle, resulting in poor overall system performance. Resource sharing in complex network systems aims at achieving maximal system performance by efficiently utilizing the system resources available.

A distinguishing feature of such complex network systems is the inability to maintain consistent global state information at distributed points of control. A complex network system can thus be viewed as a collection of distributed decision makers taking decisions to achieve common goals under uncertain local and partial views of the system state. Network latencies further complicate these systems.

Complex network systems present serious difficulties in security, heterogeneity and operability, and in performance. In this paper special concern is devoted to the last two. Our aim is to first ensure that the algorithm for adaptive resource sharing is feasible. It then, must continue to be effective and stable as the system grows. Scalable solutions are required in order to effectively utilize the aggregate processing power available and hide latencies. Such solutions strive to conceal physical characteristics such as system size, network topology, faults, and different sorts of latencies and resource capacity constraints.

PVM (Parallel Virtual Machine) is a parallel and distributed computing environment used worldwide. It enables a collection of heterogeneous computers connected by dissimilar networks to be used as a coherent and flexible concurrent computational resource. In [12] we describe a resource management system which provides a replacement for the round-robin policy with a scalable and adaptive algorithm for resource sharing [13]. This resource manager is extended in this paper to further improve performance.

The remainder of this paper is organized as follows. Resource sharing in a complex network environment is described in Section 2. Section 3 details an algorithm for preservation of mutual interests together with enhancements required to improve locality in a complex network environment. PVM and its process control are briefly described in Section 4 followed by an implementation of a RM which replaces the default PVM round-robin assignment. Initial prototype implementation measurements results are given in Section 5. Finally, conclusions derived from this study and directions recommended for future research are given in Section 6.

2 Adaptive Resource Sharing

2.1 Introduction

The problem of resource sharing was extensively studied by Distributed Systems (DS) (i.e.,[16]) and Distributed Artificial Intelligence (DAI) (i.e.,[17]) researchers, particularly in relation to the load sharing problem in such systems. Distributed systems require some mechanism for cooperation among the processors, attempting to assure that no processor is idle while there are tasks waiting for service. Similarly, a solution to the resource access problem attempts to ensure that there are no such resources idling while requests are queued at other nodes. A mutual relation between two nodes in a distributed system is identified, where one node is a service provider and the other is a service user. One of the strengths of large-scale distributed systems is that when a node cannot fulfill a service request locally, it may issue a request to another node, where the service is available. The challenge is to find a service provider and to keep the system stable and scalable while doing so. Load sharing algorithms provide an example of the cooperation mechanism required when using the mutual interest relation. A location policy determines the approach used for locating a remote resource. Information propagation, request acceptance and process transfer policies are other components of such algorithms. When incorrect information can be detected and recovered, decisions can be based on weakly consistent information which may be inaccurate at times ([7][13][18]). Weak-consistency allows inaccuracy as well as partiality. State information can be used as a hint for decision making enabling local decisions. Such state information is less expensive to maintain. The use of partial system view reduces message traffic, as less nodes are involved in any negotiation. For the benefit of maximal performance a hint should nearly always be correct.

Adaptive algorithms adjust their behavior to the dynamic state of the system. Thus, they are able to better approach the full computational power of the system. But, they also carry a lot of overhead. Their behavior might become unpredictable when faced with inaccurate information. Therefore, the complexity of the algorithms should be kept as low as possible while still allowing for a significant performance improvement. Few such algorithms are subsequently described.

2.2 Early Study

In a study by [7] the performance of location policies with different complexity levels is compared. The research was performed on load sharing algorithms. Three location policies were studied: random policy (which is not adaptive), threshold policy, and shortest policy. Random selection, which is the simplest, yields significant performance improvements in comparison with the no cooperation case. Still, a lot of excessive overhead is required for the remote execution attempts, many of which may prove to be fruit-

less. Threshold probes a limited number of nodes. It terminates the probing as soon as it finds a node with a queue length shorter than the threshold. Threshold results in a substantial further performance improvement. Shortest probes several nodes and then selects the one having the shortest queue, from among those having queue lengths shorter than the threshold. By probing nodes before actually sending a task for remote execution, the amount of data, carried by the communication network, is decreased. However, there is no added value to looking for the best solution. A node should not look for the best solution but rather an adequate one. It may thus be concluded that advanced algorithms do not necessarily entail a dramatic improvement in performance. Many approaches suggested later are based on [7] which illustrates a great advantage because of its simplicity. Its drawback lies in having to initiate negotiation with remote nodes upon request. This may result in lengthy delays. To avoid such remote message exchange state information regarding other nodes in the system should be maintained locally. In [19] such state information is held locally and periodically updated. A node often deletes information regarding resource holders which are still of interest. In order to better support similar and repeated resource access requests, cache entries of mutual interest should be retained as long as they are of interest. Such an algorithm is described next.

2.3 Flexible Load Sharing

In the Flexible Load Sharing algorithm (FLS) [13] a location policy similar to Threshold is used. In contrast to Threshold, FLS bases its decisions on local information which is possibly replicated [1] at multiple nodes. For scalability, FLS divides a system into small subsets which may overlap. Each of these subsets forms a cache (Fig.1) held at a node. Cache members are nodes of mutual interest which are first discovered by (pure) random selection. Biased random selection is used from then on in order to retain entries of mutual interest and select others to replace discarded entries. The cache actually defines a subset of system nodes, within which the node seeks a partner. That way the search scope is constrained, no matter how large the system is as a whole. The algorithm supports mutual inclusion and exclusion, and is further rendered failsafe by treating cached data as hints. It can be compared to unbiased random selection where new nodes to be included in the cache of a node are selected periodically and randomly, even if nodes sharing mutual interests existed in the current cache. In order to minimize state transfer activity, the choice is biased and nodes sharing mutual interests are retained. In this manner premature deletion is avoided. In addition, it is important to note that FLS does not attempt to produce the best possible solution, but like Threshold, it offers instead an adequate one, at a fraction of the cost. By doing so, the extra (communication and processing) overhead is saved. However, FLS presents an added value to the algorithms discussed in [7] research: the necessary information for matching partners, sharing a mutual interest, is maintained and updated locally on a regular basis, rather than waiting for the need to perform the matching to actually arise in order to start gathering the relevant information. Cache entries of mutual interest are retained as long as they are of interest. Premature deletion is thus avoided. This policy shortens the time period that passes between issuing the request for matching and actually finding a partner having a mutual interest. The FLS algorithm can be extended to any other matching problem in a distributed system, as will be shortly demonstrated.

2.4 Locality Study

Another interesting approach for locating resources is described in [14]. A local data structure which is efficiently maintained is described where state information is held of all other nodes and not only those currently of mutual interest. Again this study was in the area of load sharing. The motivation is to improve the probability that remote requests would be directed towards nodes that share a mutual interest, thus lowering the cost of the search for a mutual partner. A disadvantage of this scheme is its dependency on system size which violates scalability. In the next section we describe in detail a scalable and adaptive algorithm preserving mutual interests [13]. We then propose how to enhance locality in line with [14] within this framework.

3 Preserving Mutual Interests for Resource Sharing in Complex Network Systems

In this section we describe in more detail the flexible load sharing algorithm (FLS) [13] which supports scalable and adaptive initial allocation of processes to processors. To enable analysis of this complex environment an idealized environment (basic case) is initially assumed characterized by the following:

- no message loss
- non-negligible (>0) but constrained latencies for accessing any node from any other node
- availability of unlimited resource capacity, i.e., the number of nodes in a cache is not limited
- the selection of new resource providers to be included in the cache is not a costly operation and need not be constrained.

A local cache is maintained at each node. The main cache parameter is :

- n, the number of nodes to try to include in the cache

A cache contains the identification of other nodes with mutual interest currently known to the node together with their current state. This is discussed next.

3.1 State Metric

The algorithm for preserving mutual interests defines three states for the nodes of the system positive, negative, and neutral. with the following semantics. A negative node experiences resource shortage; a positive node has surplus resource capacity, i.e. it is capable of serving external resource access requests; a neutral node does not participate in resource sharing. A node is of interest to another node if and only if these two are of 'opposite' states. This relation is symmetric and each is interested in having the other in its cache. Therefore, a node need retain only such nodes in its cache. The state may be determined by applying thresholds to a resource state. Assume the existence of positive and negative thresholds and and let r represent the load imposed on a resource at node i. The load may be expressed as the length of the resource request queue at the node, resource utilization, a combination of the two or any other relevant measure applicable. A node resource state r is thus mapped into one of three possible states:

$$\textbf{state}\, r = \begin{cases} \textit{positive} & \textit{if} \quad R > T_+ \\ \textit{negative} & \textit{if} \quad R < T_- \\ \textit{neutral} & \textit{if} \quad T_- \leq R \leq T_+ \end{cases}$$

where R is the load measured. Scale is recognized as a primary factor influencing the design, implementation and performance of complex network systems [15]. Another scalability factor of major importance is latency (or delay). Network delays may be lengthy and also subject to high variability [11].

Note that even under the optimistic assumptions made, it is desired to constrain the scope of operation and resultant overhead. To achieve this goal, the algorithm takes advantage of weak-consistency which permits both partiality and temporary inaccuracy. Partiality is dealt with first. By constraining the view a node has of the system to a small subset of it held locally in a cache, a node considerably bounds interaction with other nodes. A node exchanges state information with all nodes in its cache and selects from these for possible remote resource access. Cache size is much smaller than system-size. Cache members include a few other nodes equipped with the resource in question [1]. Cache access time is much faster than any remote resource access. Having such information available locally means that lengthy negotiation with the name server or other nodes before a resource location is found can be avoided. Also similar and repeated requests are made more efficient.

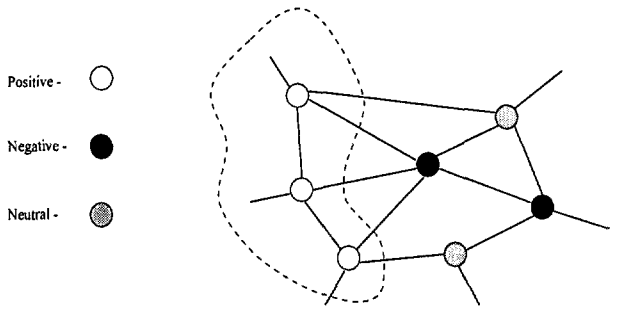


Figure 1: Resource states. A node's cache (view of the system).

An algorithm for preserving mutual interests is presented with the main aim of hiding scale. We first present a simplified form of the algorithm for preserving mutual interests. We then extend the basic algorithm to enhance locality.

3.2 Basic Algorithm for Preserving Mutual Interests

For preserving mutual interests the following location and acceptance policies are used. If the cache of a node is not empty, then a remote resource is selected from it. A request to access the resource is sent with the command piggybacked so as to allow for immediate access if the request is accepted. Otherwise a negative acknowledgment is returned. The remote resource holder employs the following acceptance policy. If the resource is available and capable of granting external requests, a positive acknowledgment is sent to the request originator; otherwise a negative acknowledgment is sent. Note that as a cache usually holds information regarding several remote locations and as the hit-ratio is high only few requests are expected to be rejected. These locations are treated as a hint [18] for decision making and incorrect decisions will be terminated rapidly. The information policy is performed upon a change in a node's local load state. This is in preference to periodic update which can be hard to tune. All cache members are then notified. A remote resource access decision is thus applied within a node, independently of its application in other nodes. Caches may overlap. Cache membership is symmetric. This symmetry is a key property for ensuring that a node is kept informed of the states of the nodes in its

[1] If such information is not available locally, a name service could be consulted to obtain other node identifiers. The latter relate to the local sub-network and also neighboring ones.

cache. A node retains useful nodes in its cache and discards nodes which are no longer of interest. Cache membership is thus dynamic and adaptive, aimed at retaining those nodes of interest and discarding others.

Two events may cause a change to the cache contents: cache refresh and message receipt described next. In the following discussion:

- C_i denotes the current cache for resource r at node i
- $|C_i|$ is the size of the cache at i
- state_i denotes the current state of node i
- $\text{mutual}_{i,s}$ is true if nodes i and s are of 'opposite' states, false otherwise. i.e.:

$$mutual_{i,s} \equiv (state_i = positive\ and\ state_s = negative)$$

$$or\ (state_i = negative\ and\ state_s = positive)$$

$$\equiv not\ (state_i = neutral\ or\ state_s = neutral\ or$$

$$state_i = state_s)$$

Obviously, this predicate on nodes i and s is symmetric and hence :

$$mutual_{i,s} = mutual_{s,i}$$

Parts of the algorithm are given as rules of the form "guard $\geq$ action" which form alternatives in a guarded command if...[]...[]... fi. Two events may cause a change to the cache contents: cache refresh and message receipt (described next).

Message Receipt - When a state_s message arrives at i from node s, the following message receipt procedure is invoked :

```
if      ( s∉Ci and mutual_{i,s}) ⇒
        insert s entry < s, state_s > into Ci;
        send a responding message state_i to node s;
   []   (s∈Ci and mutual_{i,s}) ⇒
        update s entry < s, state_s > in Ci;
   []   ( s∈Ci and not mutual_{i,s}) ⇒
        discard s entry from Ci;
   []   (s ∉Ci and not mutual_{i,s}) ⇒
        skip (ignore);

fi
```

Thus, nodes with mutual interest are included and updated while others are discarded. Hence, C_i contains only nodes of mutual interest. The response in the first case, where s is a newly selected node of mutual interest for i, acts to confirm to s that i is of interest for s. The second case ensures that s is retained if it is still of interest, but terminates the state exchange between i and s by giving no further response. The other two cases ensure that nodes with no mutual interest are discarded and ignored, respectively.

Cache Refresh - The current cache is refreshed upon initialization (the cache is initially empty), and following a resource state change[2]. Node i uses the following procedure:

```
{for all nodes k ∈ Ci}:
(disseminate state_i to node k;
            ;discard k)
            if (state_i=positive or state_i=negative)⇒
randomly select n nodes {j1...jn} from the set
{1,2,...,system_size};
            {for all nodes k∈{j1...jn} :
            disseminate state_i to node k};
            [] (state_i=neutral)⇒ skip;

            fi
```

Extensions to achieve enhanced locality are described next.

3.3 Extensions to achieve enhanced locality

Following a state change at a node, the local information about other nodes that previously had a common interest with the node is lost. This may not always be plausible. Consider a node that experiences a service surplus. It may start providing the available service to some other remote nodes. As the remote requests continue to arrive at it, it may experience a service shortage, thus changing its state. But then, as it notifies the other nodes it is no longer available for their requests and the remote requests stop. It would return to its previous, surplus, state within a short while. This scenario demonstrates how state swings may occur.

Upon returning to the surplus state, the node needs to discover nodes with a shortage for that service. In the original algorithm this search is done from scratch: nodes are probed randomly, in the hope that some of them share a common interest with the local node. A better strategy may be to save aside a cache that became invalid once the local node changed states. Then, when the node returns to the previous state, and especially in case the return to the state occurred within a short while, it would be plausible to first probe the nodes that composed the old cache, which was saved aside. Those nodes which shared a common interest with the node before it changed its state, and therefore it is likely they would share this common interest with it again, once it returns to that state. Directing the probing towards those nodes may improve performance, compared to an entirely random probing,

[2]Note that, since a cache only contains nodes of mutual interest, it will be empty immediately following a state change and consequent cache refresh. However, if the state of the node is not N, the responding messages will quickly re-establish cache membership. If the state is N, its cache will remain empty.

which is a total 'shooting in the dark'.

Hence, our improved FLS acts as follows: Two caches are used to save aside data that is no longer of interest to the local node: an old-positive-state-cache and an old-negative-state-cache. When a node changes its state from negative to some other state (neutral or positive), we clean the content of the old-negative-state, and insert to it the nodes of the current cache, before refreshing it. Likewise, when a node is in a positive state, and its state changes, the content of the current cache is copied to the old-positive-state-cache, to be saved until that information is needed once again. When a node state becomes positive (or negative), and a refresh cache is in order, the nodes in the old-positive-state-cache (or the old-negative-state-cache) are probed first. Once the old cache nodes are exhausted, probing is done at random, for a total of n probed nodes, similar to the way it is done in the original FLS algorithm.

Notice that there is no saving of cache when the node changes state from a neutral state, and there is no modification to the way cache refresh is done once a node becomes neutral. This is due to the fact that a neutral node share no common interest with any other node (this is, in fact, the way a neutral state is defined). Thus, the cache of a node in a neutral node is always empty, and there is no sense in probing other nodes as long as the local node is neutral. Our experiments are conducted in PVM is summarized next.

4 Parallel Virtual Machine

PVM is composed of two parts - the library of PVM interface routines, called "pvmlib", and the support software system. The latter is called "daemon" - pvmd and executed on all the computers making up the virtual machine. These pvmds are interconnected with each other by a network. Each daemon is responsible for all the application component processes executing on its host. There is a master daemon which controls the physical configuration and acts as a name server. Otherwise, the control of the virtual machine is completely distributed. Process control is addressed in the following paragraphs. PVM [4][8][9][10] process control includes the policies and means by which PVM manages the assignment of tasks (processes in the PVM system) to processors and controls their execution (spawning) . The computational resources may be accessed by tasks using the following policies: default (transparent) policy, architecture dependent policy, machine specific or a policy defined by the user to substitute the default (round-robin) PVM process control. In the case of default/transparent, the next node is selected from this pool in a round-robin manner. The main drawbacks of such policy are the fact that PVM ignores the load variations among the different nodes and also PVM is incapable of distinguishing between machines of different speeds. We extended PVM to provide an alternative spawn (execution) service which is scalable and adaptive [12]. This was further extended to enhance locality. Initial Performance Measurement results in PVM are described next.

5 Initial Performance Measurement Results

We experimented on a system composed of eight PentiumII based workstations that are connected by a Fast-Ethernet LAN. Each workstation is an independent source of load (each workstation generates sequence of 100 processes). PVM is the distribution environment used. The average load imposed on the system was 70 %. To measure performance in a cluster environment we measured the following three cases :

- PVM (default round-robin process allocation).
- Extended PVM. This system includes Resource Manager (RM) [12] which uses resource sharing service (FLS) to improve process allocation.
- Extended PVM with enhanced locality.

A hit is defined as a lookup request, which could be answered with the current contents of the cache. We define a miss to be a lookup request which could not be answered with the current contents of the cache, such as an overloaded node not finding an entry for an underloaded node in the cache. Cache hit should be maximized. Cases of cache miss should naturally be minimized, although to a much lesser extent. Zhou [19] has found that even when 50-70 % of the components are not eligible for remote execution, load sharing can still be beneficial.

The percentage of probing based on previous caches is defined to be the percentage of probed nodes which were chosen based on their appearance on previous caches, out of all probed nodes. Notice that as this percentage increases, the percentage of successful probing, that is probes that result in finding a node which indeed shares a common interest with the local node, is also increased. Hence, our first goal, to better direct the probing towards nodes that are more likely to share a common interest with the local node, assuming that this will increase the number of successful probes, was indeed realized.

As the probing process becomes more efficient, the cache is updated within a shorter while, when a node changes it state. A faster update of the cache means it is empty for shorter periods of time. As a consequence the miss percentage decreases. Our results verify this phenomena: The percentage of missed when the extended FLS is used, is lower than that percentage for the original FLS.

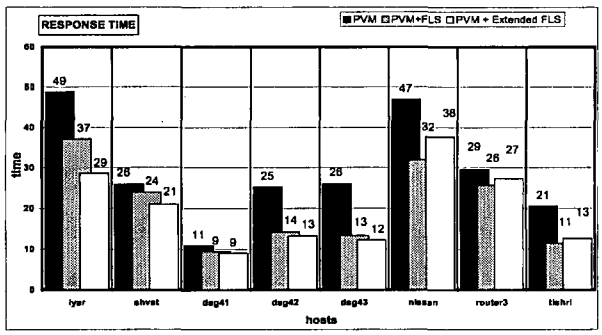


Figure 2: Response time benchmark. Eight PentiumII nodes. Each node generates 100 tasks.

It should be noted that the most noticeable decrease in the miss percentage, in our experiment, is at nodes where the percentage of probing based on previous nodes is relatively low. This seems to be a paradox at a first glance, but it is not: The nodes having the higher percentage of probing based on previous caches, in our experiment, are the two underloaded nodes. Those nodes change their state extensively due to receiving jobs from the overloaded nodes. When each of them returns to the underloaded state, it uses the cache that was previously valid for this state, to first probe the nodes that were previously found to be overloaded (and hence share a common interest with the underloaded node). Those probed nodes are, with high probability, the two overloaded (on average) nodes, and will most likely indeed be overloaded. Hence, the underloaded nodes update the overloaded nodes about their return to the underloaded state within a short while. The caches of the overloaded nodes is updated more quickly, and so it is less likely that the caches will be found to be empty when a remote execution is in order. Thus, the percentage of misses is decreased for the overloaded nodes.

To conclude: Our experiment showed that probing based on previous cache members yielded a more successful and efficient probing process. As a result caches are updated faster, and are empty less frequently. The miss ratio is thus decreased, and the number of remote execution possible to be carried out is increased. In our example (Fig.2) the extended FLS gives an improvement of 31 % in comparison to PVM. We expect substantial improvement in complex environments like internet.

6 Conclusions and Future Study

This paper presented a PVM-based implementation of an enhanced scalable and adaptive resource sharing facility. Our system is based on commodity hardware (PCs and networking) and software (PVM) offering a low cost solution as an alternative to mainframes and MPP's. Such a system adapts to state changes which are unpredictable in a complex network environment. Simulation [11] and prototype implementation [12] results demonstrate the utility of an algorithm preserving mutual interests to such environments. This was subsequently enhanced to optimize locality as described in this paper. We are encouraged by the relative ease of FLS algorithm implementation as resource sharing service and its extension and the results it provides. An extensive performance measurement study of locality is planned.

Our current implementation supports scalable and adaptive initial placement. It will be complemented by migration after start-up [2] to support a general purpose PVM-based high performance computation server. We are working on adaptation of this cluster computation server to the Internet environment and its technologies (like JAVA, CORBA). Initial results demonstrate generality of the algorithm preserving mutual interests and usefulness to support load sharing and also the dynamic parking assignment problem'. We are currently working on its customization to e-commerce.

Acknowledgments

We gratefully acknowledge the Ministry of Science grant no. 8500 for its financial support. We extend our thanks to Peggy Weinreich for her valuable editorial work.

References

[1] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. Enhancing the Web's Infrastructure: From Caching to Replication. IEEE Internet Computing, vol. 1(2), March - April 1997.

[2] A.Barak, A. Braverman, I.Gilderman, O. Laadan. Performance of PVM and the MOSIX Preemptive Process migration Scheme. Proc. 7th Israeli Conf. on Computer Systems and Software Engineering, IEEE Computer Society Press, 1996.

[3] A. Barak, S. Guday and R.G. Wheeler. The MOSIX Distributed Operating System, Load Balancing for UNIX". Lecture Notes in Computer Science, vol. 672, Springer-Verlag, 1993.

[4] A. Beguelin, J. Dongarra, G.A. Geist, W. Jiang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Mchine, a User's Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, 1994.

[5] S. Crane and K. Twidle. Constructing Distributed Unix Utilities in Regis. Proceedings of the Second International Workshop on Configurable Distributed Systems, March 1994.

[6] A. Dupuy and J. Schwartz. Nest: Network Simulation Tool. Technical Report, Communications of the ACM, vol. 33(10), October 1990.

[7] D.L. Eager, E. D. Lazowska and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. IEEE Trans. on Software Eng., vol. 12(5), pages 662-675, May 1986.

[8] G.A. Geist, J.A. Kohl, R. Manchek and P. M. Papadopoulus. New Features of PVM 3.4. 1995 EuroPVM User's Group Meeting, Lyon, France, September 1995.

[9] G.A. Geist, J.A. Kohl, P. M. Papadopoulus. "CUMULVS": Providing Fault Tolerance, Visualization and Steering of Parallel Applications. SIAM, August 1996.

[10] G.A. Geist, J.A.Kohl, P.M. Papadopoulos and S.L. Scott .Beyond PVM 3.4: What we have learned, What's next, and Why. Oak Ridge National Laboratory, Computer Science and Mathematics Division, Oak Ridge, URL http://www.epml.gov, 1997.

[11] M. Kapelevich and O. Kremien. Scalable Resource Scheduling : Design , Assessment, Prototyping. Proceedings 8th Israeli Conf. on Computer Systems and Software Engineering, IEEE Computer Society Press, 1997.

[12] M. Kemelmakher and O. Kremien. Scalable and Adaptive Resource Sharing in PVM. LNCS Proceedings, vol. 1479, pp. 196-205, Springer-Verlag, ISBN 3-540-65041-5, 1998.

[13] O. Kremien, J. Kramer and J.Magee. Scalable, Adaptive Load Sharing Algorithms. IEEE Parallel and Distributed Technology, pages 62-70, August 1993.

[14] F. Krueger, N. Shivaratri. Adaptive Location Policies for Global Scheduling. IEEE Transactions on Software Engineering, vol. 20(6), pages 432-444, June 1994.

[15] M. Satyanarayanan. Scale and Performance in Distributed File System. IEEE Transactions on Software Engineering, vol. 18(1), pages 1-8, January 1992.

[16] N. Shivaratri, P. Krueger and M. Singhal. Load Distributing for Locally Distributed Systems. IEEE Computer, pages 33-44, December 1992.

[17] R. G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. IEEE Transactions on Computers, C-29(12), pages 1104-1113, December 1980.

[18] D.B.Terry. Caching Hints in Distributed Systems. IEEE Transactions on Software Engeneering, vol. SE-13(1), pages 48-54, January 1987.

[19] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. IEEE Transactions on Software Engineering, vol. 14(9), pages 1327-1341, September 1988.

A Dynamic Load Balancing Method On A Heterogeneous Cluster Of Workstations

Alessandro Bevilacqua
Department of Physics, University of Bologna and INFN Bologna, Viale B. Pichat, 6/2, Bologna, Italy
Phone: +39 051 6305 163, Fax: +39 051 6305 047
E-mail: bevila@bo.infn.it

Keywords: dynamic load balancing, cluster of workstations, data parallelism, PVM, pool-based method, manager-workers

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 10, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

The efficient usage of workstations clusters depends first of all on the distribution of the workload. The following paper introduces a method to obtain efficient load balancing for data parallel applications through dynamic data assignment and a simple priority mechanism, on a heterogeneous cluster of workstations, assuming no prior knowledge about the workload. This model improves the performance of load balancing methods in which one or more control processes remain idle for an extended period of time. In order to investigate the performance of this method we take into consideration a problem of 3D image reconstruction that arises from events detected by a data acquisition system. Studies of our load balancing model are performed under slight and heavy load condition. Experimental results demonstrate that this model yields a substantial load balance, even more if workstations are heavily loaded, from exploiting the idle time of one control process. In addition, this strategy reduces the overhead due to communication so that it could be successfully employed in other dynamic balancing approaches.

1 Introduction

In academic and industrial institutions inexpensive clusters of workstations or PCs are replacing single, more expensive, parallel machines and small symmetric multiprocessor (SMP) systems can be found within many of the powerful modern systems. Research centers like CERN [1] are developing experiments based on clusters of thousands of LAN workstations connected to a WAN[1].

Usually, in a LAN there are connected workstations with varied performance levels, as well as very different loads and variable communication times. In addition to the heterogeneous hardware, the heterogeneity of such MIMD systems is due to their multiuser environment that makes the workload change continuously. To maximize the performance of these loosely coupled parallel systems, it is essential to minimize the idle time for each process and ensure the balancing of processes workload. The techniques involved in load balancing and task scheduling play the key roles in achieving an equal share of total workload for each process and help to minimize the total execution time[2, 3, 4].

There are many studies dealing with problems of load balancing for distributed memory systems. Some works[5, 6] assume that the processors employed are continuously lightly loaded, but commonly the load on a workstation varies in an unpredictable manner. Other studies based on data migration introduce a large amount of communication overhead[7].

This paper presents a load balancing method for data parallel applications, based on dynamic data assignment. It is suitable for a cluster of workstations even if the load changes dynamically. This method is based on a modified manager-workers model and achieves workload balancing by maximizing the useful CPU time for *all* the processes involved, even for the manager, without introducing any significant overhead due to the method itself. In addition, it can reduce the communication time needed for distributing data and for collecting results. This model is accomplished by means of a fixed *working* process, the manager, which holds all the available work and satisfies, by means of a priority queue, *idle processes*, the workers, asking it for more work.

This paper consists of 7 parts. In Section 2 we present an overview of the load balancing problem. In Section 3 we outline the image reconstruction problem used to test our load balancing model and the sequential algorithm that had been previously accomplished. Section 4 describes the load balancing strategy, the parallelized version of the previous algorithm and the priority mechanism. Section 5 contains the results of the experiments and an analysis of performance. Concluding remarks follow in Section 6. Finally, in Section

[1] The European Laboratory for Particle Physics.

7 we suggest future studies and application fields.

2 Review

To achieve the best performance from a parallel application in a heterogeneous computing environment, it is important to minimize the idle time of processes and to ensure that the workload stays evenly distributed so that each process ends its task at almost the same moment.

In a cluster of workstations the continuously changing workload makes the estimation of execution time unpredictable. Works in [5, 6] show how to achieve a good load balancing by assuming that all processors of the workstations employed are idle or lightly loaded during the whole computational period, but this is not realistic in a network of workstations. In [7] load balancing is accomplished by dynamically changing the number of virtual processors that each physical processor has to emulate, but this high granularity approach produces a large communication overhead. The load balancing method considered in [8] is designed for non uniform iterative algorithm and redistributes data after each iteration to balance the workload among all processors. However, no re-balance is considered during each iteration, that might take a long time. Moreover, in the presence of a high number of short time iterations the communication overhead becomes large. The work presented in [9] is based on a manager-workers model and exploits data redistribution for achieving load balancing. The manager directs the workers to shift data among themselves to obtain a more balanced condition, even within an iteration, but the goodness of this method is strongly dependent from theirs image processing applications. Moreover, even if that load balancing scheme tries to minimize communication costs, under certain conditions it may introduce a large overhead due to communication and it does not minimize the idle time of processes anyhow.

Our load balancing scheme gives a method to minimize the idle time of *each* processor involved and to reduce the communication overhead, by increasing data locality. In addition, it could be applied to already existing load balancing methods.

3 Image reconstruction problem and sequential algorithm

We give here a short description of this *image reconstruction problem* and of the acquisition system we used to test the performance of our load balancing model.

Positron Emission Tomography (PET) is a tomographic method[10] that allows imaging of any body that has been previously injected. The image arises from the measurement of the radiopharmaceutical distribution inside the body. PET imaging consists of two steps. First, the data acquisition system detects the number of pairs of opposite photons (*events*, derived from positrons annihilation); then, through a reconstruction method, we can obtain the original distribution, hence the related image.

In this test, the detector system is an experimental apparatus[11] made of two pairs of detectors positioned on a rotating gantry at 90 degrees from each other. After filling a phantom[12] with a radiopharmaceutic, the emission was measured. The tomograph rotated on a radius of 76mm by 90 degrees in 20 discrete steps around the phantom and detected a total amount of 10^7 events. The program that manages this acquisition system periodically stores on a file a block of 10000 events, 18 bytes an event, and it adds control data such as the time employed for the acquisition and the angle of rotation. This file will be the input file for the application. The processed events range from 70 to 90 percent of the total number of events, for statistical, physical and electronic reasons. This non-homogeneous data processing represents another cause of load unbalancing.

To solve the problem of image reconstruction we accomplished a 3D "weighted" backprojection method[11].

The phantom to be reconstructed lays into a volume of 40x40x40mm^3 represented by a 3D matrix M of dimension N^3, where N depends on the resolution of the final reconstructed *image* (here N=128). Let the *voxel* be a matrix element, and the *event line* be the ideal line joining the detectors, along which a pair of photons travels in opposite directions. Of course, the other details involved in this method are not described here because they would deviate attention from the purpose of this study. Fundamentally, this method consists of filling with weighted values w, for any accepted event, all voxels crossed by the event line. It is important to note that the amount of intersected voxels changes as considering varied event lines; thus producing a different computational load for each event.

Here briefly outlined the kernel of the sequential algorithm, in a pseudo-code:

```
1. while (there are blocks in the file) do
2.   READ one data block
3.   for (each accepted event) do
4.     for (each crossed voxel) do
5.       for (x from 1 to r) do
6.         M(i_x, j_x, k_x) = M(i_x, j_x, k_x) + w
7. WRITE M
```

where r depends on the resolution of the reconstructed *object*.

Even if a 2 bytes *integer* matrix expresses the fi-

nal volume, M is a 4 bytes *float* type matrix during the reconstruction steps. It requires 8MB that are the main memory requirements of this algorithm. As seen, there are no data dependencies among blocks and if we duplicate M on each process memory, then the *reconstruction* procedure between code lines 3-6 can be executed independently.

4 Load balancing model

4.1 Load balancing strategy

Dynamic load balancing methods can be divided into two main categories[3]. In *peer-based* methods, the work is initially distributed among different processes and among them data migration is required to achieve load balancing. In *pool-based* methods, one process has the whole work and idle processes ask it in order to get more work[13].

To investigate a method for extracting maximum performance from the system *and* also from *each* process, attention is focused on pool-based methods, which introduce a smaller overhead due to interprocessor communication. Hence, in this load balancing scheme a fixed process, called *manager* process, has all the available work and idle processes, called *workers*, ask this fixed process for more work. In the canonical manager-workers scheme the workers are self-scheduled and the manager remains idle and ready to satisfy requests as they arrive; however, it performs *only* control issues, as in [9]. In the present paper, we name our model the "*working-manager model*": the manager uses its idle time to process data itself and pushes the incoming requests into a circular queue by using a FIFO strategy. During the execution the manager checks this queue with a dynamically variable frequency.

The heuristic method is simple: *when the manager does not perform any control task it must work, but it should never choose working over managing.* As it will be seen, this strategy reduces the overhead due to communication such that it could be successfully employed in a few dynamic balancing approaches affected by high communication costs. Further, we introduce a priority mechanism, which in some cases yields a significant improvement.

4.2 Parallel algorithm

As shown above for the sequential algorithm, the *reconstruction* procedure can be executed independently, making it possible to parallelize at line 1 in the code.

After the initial data have been divided among processes in a block cycle manner, each process starts working on its local data. When a worker runs out of work, it gets a block from the manager, but while each process has work to do, *also the manager works.*

To accomplish the working-manager model we follow the master-slave paradigm, in which a separate *master* program is responsible for processes (*slaves*) spawning, data assignment and collection of results. Then, we implement the SPMD model, in which one code runs on each processor, to avoid keeping apart the master tasks from the slaves ones and to consider a future porting on monotasking MPP systems (e.g. the CRAY T3E machine). The code is written in C language by using PVM libraries for message passing[14].

Here is a brief description of the kernel of the parallel algorithm. Again, by using a pseudo-code:

```
 8. while (there are blocks) do
 9.   if (master) then CALL scheduler
10.   else RECV data
11.   for (each accepted event) do
12.     if (master AND (event mod Q) = 0) then
CALL scheduler
13.     CALL compute
14.   SEND End-Of-Block signal
15. if (master) then WRITE M
```

where *event* is the sequential order number of the accepted event, Q is an integer number, RECV (blocking) and SEND (asynchronous) are PVM routines, *compute* is a *fragment* of the reconstruction method. *scheduler* has the following pseudo-code:

```
16. while (N_RECV End-Of-Block signal) do PUSH
(procnum, priority)
17. while (queue not empty OR procnum is master)
do
18.   READ data
19.   if (queue not empty) then
20.     POP (head)
21.     SEND data
```

PUSH and POP are the typical operations over queues, *procnum* is the sender process identifier, *head* is the first process in queue, N_RECV is a PVM non-blocking routine. The following sub-section deals with the *priority* mechanism.

In the sequential algorithm line 11 is located inside the *reconstruction* procedure, as it should logically be - the cycle *for* also belongs to the reconstruction step. We can observe how it is possible to break off a computing procedure for inserting a call to *scheduler*: the manager temporarily leaves its worker job to deal with its manager task.

This procedure is incorrectly called *scheduler*, meaning that the manager acts there *as if* it was a scheduler. In the algorithm, the time slice that the manager uses between two subsequent scheduling inside the computing procedure is dynamically determined

by "event mod Q", but it could be utilized any other cyclic mechanism. A disadvantage in using this dynamic timing technique could occur if a too high value for Q were considered - the manager would become too busy in worker tasks. In this case, it could be said that the manager works instead of accomplishing its prior issues. However, this is a hypothetical condition because usually the computing time needed to process an event and the scheduling time slice should differ from a few order of magnitudes.

4.3 Priority queue

Scheduling policies work in conjunction with priority levels: the higher the priority of a process, the more the process is executed. Each process begins its execution with a base priority that can change as the application runs, depending on the application requirements.

The topic of process load measurement is associated to problems dealing with load balancing, even more if the environment is a multitasking system. Many studies show how the amount of processed data can be used as workload estimation[9, 15, 16]. We use the ratio between the accepted events and the block execution time as a performance index (the *speed* of the process)[17]. The priority mechanism assigns the highest priority to the fastest process, so that it may execute more blocks.

Let i be the order number of a block computed by a process and E_i the number of accepted events for that block. Then the priority P of a process to compute the next block i+1 is its current speed:

$$P_{i+1} = \frac{E_i}{(T_e + T_c)} \quad (1)$$

It is important to recognize that in a heterogeneous system it does not mean that there is prior knowledge about *each* block execution time; it only gives us information on a block arising from the previous one. Here T_e is the execution time and T_c is the communication time needed to transfer the data block from the ending of the SEND, performed by the manager (line code 21), to the beginning of the RECV (line code 10), performed by the worker. Both timing measures refer to the *elapsed* time. Even if we make the worker pay for the manager also, it has been verified that under various load situations, for each block $T_e \gg T_c$, therefore the linear order index of the priority queue is computed by considering T_e only.

Many load balancing methods suffer due to the last task processing[18]. In fact, execution ends as the last task ends. In our scheduling approach the worst case consists of a condition in which all processes are ending except for the slowest one, which is queued and it is going to receive the last block. To avoid this, we must know when the running processes will end, their performance and a rough estimation of the last block execution time for each process.

By using certain peculiarities of the test problem and keeping statistics of all the events already processed, we can give a rough *qualitative* estimation of E_i for each one of the last n blocks, where n is the total number of processes involved, while they are processed. Thus, by means of E_i and the current speed P_{i+1} we can deduct an estimation of the block execution time T_e. Consequently, for each processing block we can approximately estimate *when* it will finish. Let RT_j be the remaining execution time of the process j (one of the last n running processes) and TL_j be its expected execution time on the last block; then, the most likely process to receive the last block is the one for which $(RT_j + TL_j)$ is minimum.

5 Performance analysis and experimental results

The cluster used consists of 4 workstations, connected to a LAN by a 100Mbit Ethernet, except for W3, that mounts a 10Mbit adapter:

W1) SMP system: 2 PII 400Mhz, 512MB;
W2) SMP system: 2 PPro 200Mhz, 128MB;
W3) AMD K6-3D, 300Mhz, 64MB;
W4) DEC AXP 4/200, 200Mhz, 256MB.

Workstations are listed according to their performance on the sequential algorithm with lightly loaded machines (i.e. W1 is the fastest one).

The operating system is Linux 2.0 for all the workstations except for AXP, that comes with its native OSF/1; PVM is the communication library and gcc the C compiler. As we can see this is a low-cost cluster, with no cost software, but OSF/1.

Most of the difficulty in studying job performance of a heterogeneous cluster arises from its multi-tasking environment, in which different tasks running on each node may change in an unpredictable way the workload of the workstations belonging to the cluster. When studying the performance of a parallel application, one must take into consideration those tasks that produce the so-called *external load* and limit the resources for our jobs.

The aim of our measurement is to evaluate the improvement introduced by forcing the manager to work, with lightly and heavily loaded workstations. In the latter case, to get more reliable results, an artificial external load is added by running on each processor the same Monte Carlo simulation, a very CPU intensive job which keeps the external load unchanged during the execution of our tasks (same results have been obtained by utilizing I/O intensive jobs). Since we are not interested in absolute measurements[18] of the

performance of *this* test application the external load is not exactly quantified, unlike other studies[17], but we keep it constant. Further, we do not quantify the communication time because this method does not introduce any extra overhead due to communication.

This test is accomplished by using 6 processes, one a processor. To get reliable performance data, 10 executions occurred for each measurement and the reported values are the averaged ones.

The focus is on the following time measures, using the Unix/Linux OS *times* routine:

- *idle time*: the mean CPU idle time, since a worker sends the End-Of-Block signal (code line 14) until it receives next data (code line 10);
- *CPU computing time*: the CPU time it takes to a process to produce its local result;
- *global execution time*: the elapsed time due to obtain the final result;
- *unbalancing time*: difference in absolute value between the global execution time and the elapsed time of the first ending process, after it has sent its result.

We give the percentage of idle time compared with the total CPU computing time and the percentage of the unbalancing time compared with the global execution time.

The overhead introduced by this method is due to the execution of the *scheduler* procedure. We kept statistics about the duration of the *scheduler* with an empty queue and it takes a CPU time ranging from $5{\cdot}10^{-6}$s for W1 slightly loaded to $0.3{\cdot}10^{-3}$s for W4 heavily loaded and an averaged value of $80{\cdot}10^{-6}$s.

In these tests we selected Q=50 (line 12). It has been empirically seen that values less than 50 do not yield any significant improvement, but when Q increases over 100 it makes the idle time increase rapidly. When considering a mean value of about 7500 accepted events, there are almost 150 calls a block. Thus, the overhead T_{lb} introduced by our load balancing scheme is:

$$T_{lb} = (150 \cdot 10^3) \cdot (80 \cdot 10^{-6} s) = 12s \qquad (2)$$

This is less than 0.5% on the average of all the CPU execution times.

In all the figures comparing the working-manager method ("M") with the canonical one, in which the manager does not work: both situations were accomplished with heavily ("H") and slightly ("S") loaded workstations.

In Fig.1 all 16 execution times are expressed in seconds with workstation Wi acting as master and in Fig.2 the improvements introduced by making the manager work. The first result consists of an improvement on each workstation *independently* from the external load; further, W4, the slowest workstation of the cluster, has greater improvement in a heavily loaded system.

As expected, the improvement is proportional to the manager capability, and it decreases with a slow host as master. W1 is a powerful dual processor system: thus, having a manager that works a lot means adding a *powerful* resource; oppositely for a slow workstation.

Fig.3 shows the trend of the ratio between the mean CPU idle time and the total CPU execution time. Here it is essential to recognize that when the manager does not work we do not take into consideration its idle time (always more than 99%) to compute the mean value (i.e. this is the workers only idle time). This might be more correct, being it also involved in the execution, and results would be *much* better for our method, here we would rather show that we exploit all the idle time of the manager by not increasing the mean idle time of the workers: our heuristic method is accomplished.

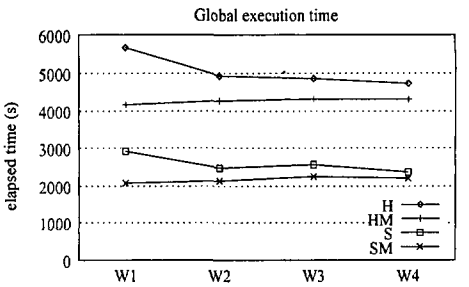


Fig. 1: global execution time with the master hosted in the workstation Wi, under heavy (H) and slight (S) load condition, with (M) and without the manager working.

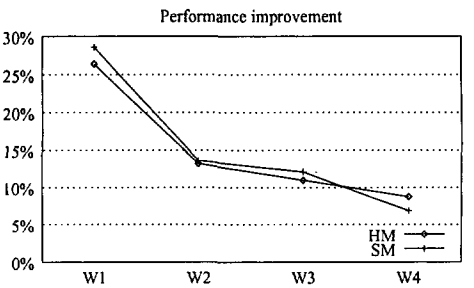


Fig. 2: performance improvement obtained by making the manager work, under both cases of heavy (HM) and slight (SM) load conditions, with the master hosted in the workstation Wi.

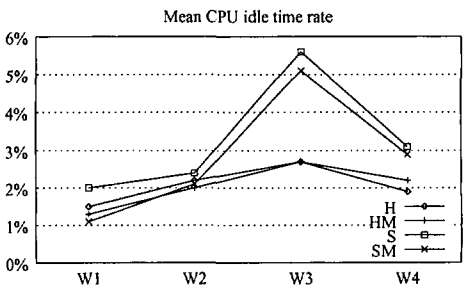


Fig. 3: percentage of the mean CPU idle time on the total CPU time with the master hosted in the workstation Wi, under

heavy (H) and slight (S) load condition, with (M) and without the manager working.

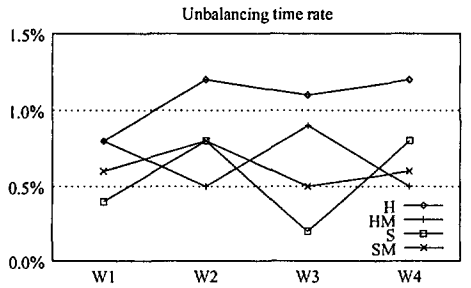


Fig. 4: percentage of unbalancing time expressed as the ratio between |TMAX - TMIN| and the elapsed time of the application, where TMIN and TMAX are the elapsed time it takes respectively to the first and to the last process to get results.

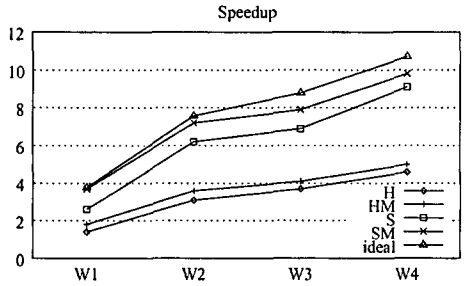


Fig. 5: speedup values S_i expressed as the ratio between T_s and T_p, respectively the time of the sequential and the parallel algorithm on W*i*.

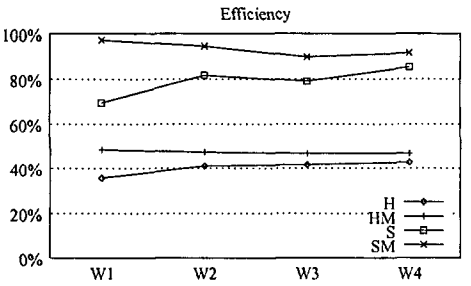


Fig. 6: "weighted" efficiency defined as the ratio between S_i and the weight (P_T/P_i) of the workstation W*i*.

In fact, in all cases under both heavy and slight load conditions, values are better when the manager works, except in the case of W4 (HM). This is more than we expected.

In this model, the manager keeps some blocks for itself: this increases data locality and reduces the communication time, thus the mean idle time that processes spend waiting for data.

In Fig.3 the anomalous behavior of W3 in slight load condition is due to the communication overhead. We must keep in mind that it has a bandwidth of one tenth compared with the bandwidth of the other workstations connected to the LAN. Lines H and HM show how the increasing of the communication latency time in all the heavily loaded workstations reduces the gap between bandwidths.

In Fig.4 it can be seen that in all the considered options the percentage of unbalancing stays under 1.2% of the total execution time. The reduction of the idle time rate obtained by making the manager work (Fig.3), under slight load condition yields a rise in the unbalancing time rate (Fig.4)(except for W4), even if in terms of "seconds" these are small differences. On the contrary, under heavy load condition latency and overhead due to the communication time increase and the working-manager model *always* improves the load balancing. The percentage of unbalancing rate ranges from 0.5 to 0.8 percent: it is more significant for W2 and W4, which have a lower unbalancing rate under heavy load condition.

The working-manager model increases data locality and cuts communication overhead: this reduces the unbalancing rate. It becomes evident in heavily loaded workstations and it comes out even from analyzing the behavior of W2 and W4 in Fig.4 (H and HM). W2 is a SMP workstation, as W1 is, and the external load consists of *two* Monte Carlo simulations. Thus, even if the load is evenly distributed between the two CPUs, the latency time affects W2 more than W1, also due to W1 100Mhz technology. Regarding W4, the manager works slowly, distributes more blocks and spends more time for communications. In addition, it is made with old technology and in heavy load condition its latency time increases.

Each one of the lines in graph Fig.4 shows that under any condition the difference between the maximum and minimum value stays less than 0.4% ; generally, due to the priority mechanism.

Fig.5 shows the speedup values S_i defined as:

$$S_i = \frac{T_s(Wi)}{T_p(Wi)} \tag{3}$$

T_s(W*i*) is the execution time it takes to the sequential algorithm on the workstation W*i*, under slight load condition, and T_p(W*i*) is the execution time of the parallel algorithm with the master hosted on the workstation W*i*. The upper line is the "ideal" speedup I_i, calculated as follows:

$$I_i = \frac{P_T}{P_i} \tag{4}$$

P_i=T_s(W1)/T_s(W*i*) is the *power weight*[19] of W*i* compared with that of the fastest workstation (W1) and P_T is the *power weight of the cluster*, obtained by summing over all P_i and keeping in mind to add twice the power weight of the SMPs. Thus we find P_T=3.78, i.e. 3.78 times of *one* W1 processor, for *this* cluster and *this* algorithm.

Finally, Fig.6 shows the "weighted" efficiency Eff_i=S_i/I_i. Under light load condition the efficiency ranges from 92 (W3 SM) to 98 (W1 SM) percent:

this means that the scheduling policies and working-manager model work fine.

This recognizes the magnitude of how much a cluster of low cost workstations can improve work production, if good algorithms run on it; as well as, by developing good balancing algorithms the hardware resources reach maximum potential, as in the 2 processors of an SMP system.

The priority mechanism, with few dozens blocks significantly contributes to decreasing the global execution time until 25-30 percent, even if it is no possible to appreciate this improvement due to the shortness of these executions. On the other hand, for long time executions the natural distribution of the running tasks limits the effectiveness of this priority mechanism. Nevertheless, it stays effective for the processing of the last block. Even more when workstations are heavily loaded, when a last long job may result in a substantial loss of balance.

6 Conclusions

It is widely recognized that a network of workstations has a lot of unused computing resources. To maximize its potential, it is necessary to write parallel algorithms that evenly distribute the workload among the workstations to maximize the load balancing and to minimize the idle time of *each* workstation involved in the computation.

This study proposes a dynamic load balancing model for data parallel applications based on a modified manager-workers paradigm: we called it *working-manager model* . This model, which exploits the idle time of the manager process to make it work, is suitable *as is* for not a large cluster of workstations. Our model has a twofold advantage, both in terms of global performance and load balancing. If the manager is hosted in the most powerful workstation of the cluster, this model achieves remarkable improvement in terms of global performance, in both cases of heavily and slightly loaded machines. Otherwise, if hosted in the "slowest" workstation, this scheme leads to a desired load balance by decreasing communications from the manager to the workers since the manager itself processes some blocks, fetching them locally. Thus this method is also suitable for clusters with high network traffic. In addition, in the case of heavily loaded machines, the working-manager method always performs a better load balancing compared with the canonical one, in which the manager does not work. In both load conditions, except when the master is hosted in the slowest workstation in the heavy load system, this model *always* decreases the ratio between the mean CPU idle time and the total CPU time.

Finally, experimental results show efficiency values for this method of over 90%, confirming that this scheduling approach exploits the whole power of each processor.

The current implementation assumes that the cluster of workstations be *one* pool of processes. Since our method exploits the idle time of a manager process, if the number of processes is heavily increased, the manager would be more continuously engaged in its control issues. Thus, reducing its idle time, and this method would loose its effectiveness. However, it had always better to split a lot of processes into different pools. In fact, trying to exploit the manager idle time by increasing its control issues (i.e., by increasing the number of workers) may lead to having more requests than the manager can satisfy. And also, this increases the time it spends for communication.

7 Future work

The cluster used could be considered as one pool of processors in a pool-based system. This achieves an efficient system-wide scheduling policy, but if the number of processes becomes large then the working-manager method looses its effectiveness.

We intend to investigate the possibility to extend the working-manager model to a much wider cluster, by dividing it into small size pools and assigning a manager to each one. In such a hierarchical structure, a working *super-manager* should treat the other managers like they treat their workers; therefore we would have a structure of many pools, each one well balanced. Within such a model, the analysis for reclustering or data migration reduces *exclusively* to monitoring the working pool managers and communications will only take place among them.

Moreover, our scheme can be applied to all the pool-based methods, and to other already existing load balancing policies, to split large clusters into smaller ones and to break down costs due to communications, if any, without loosing in terms of balancing.

Another important application can be found in the field of the real-time event processing, arising from the data sampled by an acquisition system, in which data have been arranged into blocks, as in the case we analyzed. We are already engaged in studying such a problem, by using another image reconstruction method, but keeping the same parallel structure we presented.

References

[1] (1996) Technical Proposal for CMS Computing. *CERN/LHCC 96-45*, p. 1-98.

[2] Diekmann R. & Monien B. (1997) Load Balancing Strategies for Distributed Memory Machines. *Computer Science Technical Report Series "SFB", No.*

tr-rsfb-97-050, Univ. Of Paderborn, Germany, p. 1-37.

[3] Kumar V., Grama A., Gupta A. & Karypis G. (1994) Introduction to Parallel Computing: Design and Analysis of Algorithms. *Benjamin Cummings*, New York, p. 1-597.

[4] Lüling R., Monien B. & Ramme F. (1991) A study of dynamic load balancing algorithms. *Proceedings of the 3rd IEEE SPDP*, p. 686-689.

[5] Lee C. K. & Hamdi M. (1995) Parallel image processing application on a network of workstations. *Parallel Computing*, 21, p. 137-160.

[6] Lee C. K. & Hamdi M. (1994) Efficient parallel image processing application on a network of distributed workstations. *Proc. 8th Internat. Parallel Processing Symposium*, p. 52-59.

[7] Nedeljkovic N. & Quinn M. J. (1993) Data-Parallel Programming on a Network of Heterogeneous Workstations. *Concurrency: Practice and Experience*, 5, 4, p. 257-268.

[8] Miguet S. & Robert Y. (1991) Elastic load-balancing for image processing algorithm. *Parallel Computation: Proc. 1st Internat. ACPC Conf.*, p. 438-451.

[9] Hamdi M. & Lee C. K. (1997) Dynamic load-balancing of image processing application on clusters of workstations. *Parallel Computing*, 22, p. 1477-1492.

[10] Herman G. T. (1980) Image reconstruction from projections: The foundamental of Computerized Tomography. *Academic Press*, New York.

[11] Bevilacqua A., Bollini D., Del Guerra A., Di Domenico G., Galli M., Scandola M. & Zavattini G. (1998) A 3D Monte Carlo simulation of a small animal Positron Emission Tomograph with millimeter spatial resolution. *Submitted to IEEE Transactions on Nuclear Science.*

[12] Webb S. et al. The spatial resolution of a rotating gamma camera tomographic facility. *The British Journal of Radiology*, 56, p. 939-944.

[13] Zhou S. & Brecht T. (1991) Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors. *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, p. 133-142.

[14] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R. & Sunderam V. (1994) PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing. *MIT Press*, London.

[15] Altevogt P. & Linke A. (1993) An algorithm for Dynamic Load Balancing of Synchronous Monte Carlo Simulations on Multiprocessor Systems. *IBM preprint 75.93.08, hep-lat/9310021*, Germany.

[16] Meisgen F. (1997) Dynamic Load Balancing for Simulations of Biological Aging. *International Journal of Modern Physics C*, 8, 3, p. 575-582.

[17] Schnekenburger T. & Huber M. (1994) Heterogeneous Partitioning in a Workstation Network. *8th Int. Parallel Processing Symposium, Workshop on Heterogeneous Computing, IEEE*, Cancun, Mexico, p. 72-77.

[18] Schnekenburger T. (1993) Efficiency of Parallel Programs in Multi-Tasking Environments. *PEPS Performance Evaluation of Parallel Systems*, University of Warwick, GB, p. 75-82.

[19] Meisgen F. & Speckenmeyer E. (1997) Dynamic Load Balancing on Clusters of Heterogeneous Workstations. *Report No. 97-261*, Department of Computer Science, University of Cologne, Germany.

Minimizing Communication Conflicts with Load-Skewing Task Assignment Techniques on Network of Workstations

Wei-Ming Lin and Wei Xie
Division of Engineering
The University of Texas at San Antonio
San Antonio, TX 78249, USA
Phone: (210) 458-5529, Fax: (210) 458-5589
E-mail: wlin@voyager1.utsa.edu

Keywords: Load Balance, Task Allocation, Communication Resource Conflict, Divide-and-Conquer, Network of Workstations

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 9, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

Communication latency is an important factor in deciding performance of a parallel or distributed algorithm, especially in a low speed network environment. In a bus-based network of workstations, a perfectly load balance arrangement does not always lead to the best performance due to potential communication resource conflicts. Such a situation arises when workstations tend to compete for the shared bus after they all finish their assigned workload at about the same time under such a load arrangement. In this paper, we provide a thorough analysis on how such communication conflicts can be minimized in a bus-based system by using a load-skewing assignment method. A probablistic model is used to analyze the needed skewing factor for cases in which computation requirement is either a deterministic or nondeterministic quantity. Our analytical results are closely confirmed by various simulation and experiment outcome.

1 Introduction

Network of workstations (*NOW*) is regarded as a potential parallel computing environment for its low cost and flexibility. Task partition and allocation among the workstations is a very important factor that dearly affects the system performance. Workload is usually partitioned equally, whenever feasible, among workstations to achieve the best balance from computational aspect. In a bus-based *NOW*, such a load balance arrangement does not always lead to the best possible performance due to potential conflicts on communication resources. A more in-depth analysis is needed to determine the best workload arrangement among workstations to avoid such conflicts.

There have been a great deal of research works on load balance on both homogeneous and heterogeneous systems [4, 6, 9, 14, 16, 18, 21]. Either static or dynamic task allocation strategy is discussed in these papers to reach at the best load balance among processors. The most commonly used static task allocation strategy in a distributed computing environment is the weighted task allocation method [9], which partitions task according to an estimated relative machine speed. A common dynamic scheduling scheme for distributed systems is the work stealing technique ([3],etc.) where work is dynamically migrated from heavily loaded processors to lightly loaded ones. None of these strategies takes the communication latency or potential communication resource conflicts into consideration.

Communication latency is an important factor in deciding performance of a parallel or distributed algorithm, especially in a low speed network environment or in a communication-intensive task situation. Several methods [2, 17, 20] have been devised in an attempt to compensate for the latency from the aspects of both hardware and software. But most of these proposed methods fit the well-structured unit-delay machines, such as hypercube and mesh. Although there have been various high-speed switches and communication protocols deigned to speed up communication in a *NOW*, a bus-based configuration still represents the most economic and widely available parallel computing platform. None of the known techniques discusses the effects of communication latency on task partition and allocation in such a bus-based environment. In this paper, we will study the relationship between parallel computing time and task allocation in a bus-based *NOW* by taking these communication issues into account.

Divide-and-conquer approach has been shown to be a relatively simple however widely applicable parallelization strategy in solving problems in various scientific and engineering areas. In a wide range of problems, especially in Artificial Intelligence, Computer Vision, Graph Theory, Computational Field Simulation,

Discrete Event Simulation, etc., usually when solved using a number of processors, presumably "equal" amount of subtasks are allocated to each processor. Typically at the end of the computation of each subtask, communication between the subtasks is required to reach the final result. Each such subtask can correspond to a branch of a research tree, a fixed-size block of matrix/vector elements in a sparse matrix, a fixed-size partition of input numbers in various sorting techniques, an event or a group of events in discrete event simulation, etc. A nondeterministic computation requirement is usually associated with each such subtask. Under the effect of such nondeterministic factors, if each subtask exhibits identical probabilistic behavior, which is true in most of parallel computations from divide-and-conquer approaches by "equally" partitioning the task into subtasks, the execution time of each processor can be modeled by the same probabilistic density function [15, 10]. SPMD (Single-Program-Multiple-Data) is a widely used parallel programming/execution paradigm used to solve this kind of divide-and-conquer problems. In [10], Lin and Yang have analyzed the performance of such divide-and-conquer problems disregarding communication factors.

A perfect load balance for such divide-and-conquer problems in a bus-based *NOW* will pose a high possibility of bus contention among workstations when they all reach the same communication stage. This would lead to either a simple bus conflict or even a more problematic bus collision scenario. If such a problem is not carefully addressed, an unpredictably significant amount of communication delay would take place and seriously downgrade the overall parallel performance and even offset any possible gain from such a parallel execution. Due to the nature of the shared communication medium, load assigned to workstations may have to be skewed with an amount so as to fully utilize the bus for communication. In this paper, we use a probabilistic model that assumes the computation times of subtasks assigned to the workstations obey a natural distribution. Such a model is capable of coping with the aforementioned nondeterministic nature of computation requirement. A thorough analysis is then used to determine the amount of load-skew needed in load distribution to reach the best overall performance.

The rest of this paper is organized as follows. In Section 2, problem model and our analysis approach are described. Simulation and experiment results are presented in Section 3, followed by two complete parallel programming examples in Section 4. Concluding remarks are made in the last section.

2 Model and Analysis

2.1 Problem Model

There are many scientific and engineering problems that can be approached with a divide-and-conquer parallel strategy. For example, search problems in many fields represent a perfect candidate for this due to their natural problem-solving formation as a tree search process in which the search tree can be allocated presumably "equally", in a probabilistic sense, into processors (or workstations in this study) used. Sorting problems, image processing problems, as well as many others all can be approached with such a divide-and-conquer solution technique. This problem model can be described as: "A number of processors are used to process, in an SPMD fashion, a given number of tasks, all identical in terms of probabilistic behavior in their execution times." Namely, the probability density function (*pdf*) of execution time spent in a task is identical to that of the others. Note that a *task* here is referred to a large nondeterministic number of computation steps. Any divide-and-conquer parallel algorithm solving such problems with a run-time data-dependent nondeterministic behavior by dividing the tasks presumably "equally" among a number of processors would simply lead to a result that the *pdf*s of all processors' execution times are identical. Normal distribution as well as some others has been found to be a very reliable modeling function for such execution times [12, 13, 19, 22]. In this paper, we focus on using a normal distribution function as our modeling function due to its widely found applications.

In addition to the computation model just described, this study assumes that necessary communication stages are interspersed with the computation stages. In a typical SPMD solution algorithm, these communication requirements come from, to name a few, data exchange for domain coverage, updating local solutions, combining local results into a final solution, etc. Such a communication stage usually involves all processors in the system and a similar amount of information is originated from each processor with its destination being either a controller (master processor) or some other processor(s) in the system. For example, a simple one-stage divide-and-conquer algorithm usually requires partial solutions to be gathered in a processor at the end with a communication stage after each processor finishes its assigned computation workload. As another prevalent programming example, a typical n-dimensional hypercube normal algorithm requires n computation stages with a communication stage in between two adjacent computation stages to exchange data between processors across a given dimension.

For the convenience of analysis, it is assumed that the computing environment (*NOW*) consists of a cluster of workstations with identical computing capabil-

ity connected via a communication bus, such as Ethernet. With this assumption, we can simply focus on behavior of the solution algorithms and problems without worrying about any nondeterministic factors on the computation times caused by the hardware. Note that such an assumption on processor homogeneity can be further relaxed by adopting a relative computing power factor in our analysis. We also assume here the communication protocol used between processors is message-passing, with a commonly used "nonblocking-send" and "blocking-receive" scheme. In such a scheme, a sending processor needs only send its message to the receiving buffer disregarding whether the receiving processor has reached the corresponding receiving statement in its execution; that is, the sending processor is allowed to proceed its execution as soon as it finishes such a 'send' operation (statement). On the other hand, a receiving processor when expecting a message with a 'receive' operation (statement) cannot proceed until the message is actually received in its buffer.

Since a communication bus is shared among all processors (workstations), if all processors reach the same communication stage at about the same time, bus conflicts and collisions would occur and result in undesired waiting time in processors, i.e., wasted CPU time. This problem is more likely to occur if the preceding computation stage has the workload divided "equally" among processors. Let such an assignment scheme be denoted as "Perfectly Balanced Load Assignment" (*PBLA*). Instead, if the workload assignment is skewed away from the *PBLA* by an amount such that degree of the aforementioned conflicts and collisions is reduced, then a gain in overall system performance can be expected.

- **Deterministic Case:**

Such a "Skewed Load Assignment" (*SLA*) approach can be illustrated by a simplified example as discussed in the following. A one-stage divide-and-conquer algorithm with a deterministic amount of computation requirement is to be processed on a P-processor bus-connected system. Let T_{comp} denote the deterministic overall computation time requirement, and T_{comm} denote the necessary communication time for each processor after its corresponding one-stage computation stage. With the *PBLA* approach, each processor is assigned an identical workload of $\frac{T_{comp}}{P}$ and reaches the communication stage at the same time with all other processors. Disregarding the more problematic case of bus 'collision', all processors will take turn occupying the bus for a time of T_{comm}, assuming no packet interleaving to simplify our discussion. Thus, the overall parallel time (denoted as T_{PBLA} for the *PBLA* approach) becomes

$$T_{PBLA} = \frac{T_{comp}}{P} + P \times T_{comm}$$

Instead, with an *SLA* approach, computation load can be assigned to processors such that a processor reaches its communication stage exactly at the instant when another processor finishes its corresponding communication stage. Let T_{comp_i} denote the computation load (time) assigned to processor i, where $0 \leq i \leq P-1$. Such an optimal condition can be satisfied with the following assignment pattern

$$T_{comp_{i+1}} - T_{comp_i} = T_{comm} , \quad 0 \leq i \leq P-2$$

Combined with

$$\sum_{i=0}^{P-1} T_{comp_i} = T_{comp}$$

it leads to

$$T_{comp_i} = \frac{T_{comp}}{P} + (i - \frac{P-1}{2}) \times T_{comm}$$

and the total parallel execution time (denoted as T_{SLA}) then becomes

$$T_{SLA} = T_{comp_{P-1}} + T_{comm} = \frac{T_{comp}}{P} + \frac{(P+1)}{2} \times T_{comm}$$

Comparing T_{SLA} versus T_{PBLA}, a saving of $\frac{P-1}{2} T_{comm}$ time is achieved. Note that in a message-passing communication on a bus-based system, communication latency is usually dominated by software setup times, e.g. packing/unpacking time, rather than the actual hardware communication delay, and is insignificantly affected by the length of message. Thus, it remains a good approximation by assuming an identical T_{comm} for all processors and it does not deviate much from the original T_{comm}. An example is illustrated in Figure 1 where $P = 4$, $T_{comp} = 32$, and $T_{comm} = 1$. A

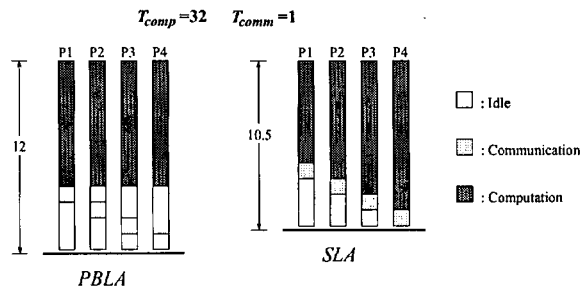


Figure 1: An Illustrating Example on Deterministic Case

PBLA approach on the left leads to a total time of 12, whereas an *SLA* on the right with a load assignment of $T_{comp_0} = 6.5$, $T_{comp_1} = 7.5$, $T_{comp_2} = 8.5$, $T_{comp_3} = 9.5$, completes the execution in a time of 10.5, representing a saving of 1.5 time units.

- **Nondeterministic Case:**

As described in our problem solution model, many solution algorithms, such as many sorting and searching techniques, an nondeterministic computation requirement is associated with each task. Such an nondeterministic factor can come from data-dependent decision-making statements in the algorithms. A different amount of load-skew than a simple T_{comm} may

be needed due to this nondeterministic nature in order to reach the best overall performance. Discussions in this paper are restricted to one-stage divide-and-conquer solution algorithms; however, our model can be extended to address the cases in which such an restriction is relaxed. Let us assume the task is partitioned into P subtasks, t_i, $0 \leq i \leq P-1$, one for each processor, in an *SLA* approach, and the computation time for task t_i, denoted as T_i, obeys the aforementioned normal distribution, i.e., $T_i \sim N(\mu_i, \sigma_i^2)$. T_{comm} denotes the communication requirement for each processor as in the previous discussion. To facilitate a simple analysis for the *SLA* approach, we further assume that the "expected" workloads are skewed among processors in a regular equi-difference fashion, i.e.,

$$\delta_\mu = \mu_{i+1} - \mu_i \ , \quad 0 \leq i \leq P-2$$

and δ_μ is called the *skewing mass*. Due to the existence of σ_i's ($\sigma_i \neq 0$ for a nondeterministic case), an optimal δ_μ value that leads to the least communication conflict is no longer a simple T_{comm} as in the deterministic case. Intuitively, it is not difficult to see that when the variances are smaller than a certain value, overall performance will still benefit from an *SLA* approach, whereas a *PBAL* approach will be sufficient when such a threshold is exceeded.

To simplify our analysis in determining the optimal skewing mass, we further assume identical σ for all subtasks' computing times. That is,

$$\sigma_0 = \sigma_1 = \cdots = \sigma_{P-1} = \sigma$$

which is not an unrealistic assumption if the skewing amount is not large. This is due to the fact that σ changes in a square-root pace of that of μ. Due to the nondeterministic nature in T_is' computation requirement, the imposed skewing mass will no longer be the actual ("effective") skewing amount between processors that finish their work in the actual sequential order. To determine the difference between these two, we let $T_{<i>}$ and $\mu_{<i>}$ respectively denote the random variable for the i-th finished subtask's computation time and its corresponding mean. Thus,

$$T_{<0>} = \min(T_0, T_1, T_2, ..., T_{P-1}) \quad (1)$$

$$T_{<P-1>} = \max(T_0, T_1, T_2, ..., T_{P-1}) \quad (2)$$

Let the *effective skewing mass* be denoted as δ_μ^e and following a similar "equi-difference" assumption on $\mu_{<i>}$'s

$$\delta_\mu^e = \mu_{<i+1>} - \mu_{<i>} \ , \quad 0 \leq i \leq P-2$$

Such an assumption is made for the sake of simplifying our analysis, and it is found to be a rather accurate one when δ_μ is small relative to μ_i's. If we can ensure that

$$\delta_\mu^e = T_{comm} \ , \quad 0 \leq i \leq P-2$$

then optimal overall parallel performance is achieved. Figure 2 shows a qualitative comparison between these

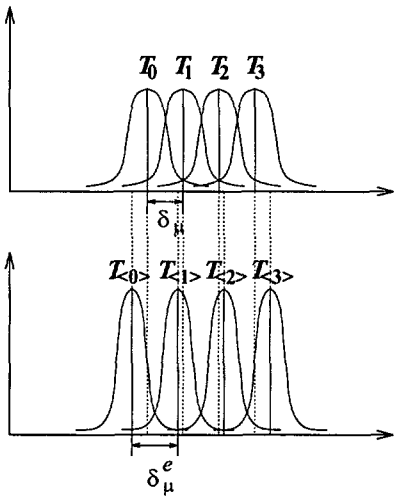


Figure 2: An Illustrative Comparison of *pdf*s

*pdf*s. The figure on the top shows the *pdf*s of desired allocation pattern with an amount of skewing mass set to δ_μ (which is yet to be determined), whereas the bottom figure displays a 'statistical' result after the actual sequential order among processors in finishing their allocated tasks is found. Note that, with a nonzero σ, $\mu_{<0>}$ is pushed to the left compared to μ_0 and $\mu_{<P-1>}$ is to the right of μ_{P-1}. Thus, δ_μ is expected to be smaller than δ_μ^e (and T_{comm}) when $\sigma \neq 0$ since

$$\delta_\mu = \frac{\mu_{P-1} - \mu_0}{P-1} < \frac{\mu_{<P-1>} - \mu_{<0>}}{P-1} = \delta_\mu^e$$

2.2 Analysis

To derive an optimal δ_μ requires a very complicated multiple integration process. We here choose to find a bound based on a much simpler analysis.

First of all, from the above discussion, we have

$$\mu_{<P-1>} - \mu_{<0>} = (P-1)T_{comm} \quad (3)$$

With the assumption that T_i's are independent normally distributed random variables ($T_i \sim N(\mu_i, \sigma_i^2)$), $\mu_{<P-1>}$ and $\mu_{<0>}$ can be expressed as ([15])

$$\begin{aligned} \mu_{<P-1>} &= E\{\max(T_0, T_1, ..., T_{P-1})\} \\ &= \int_{-\infty}^{\infty} [S(x) - \prod_{i=0}^{P-1} F_{T_i}(x)] dx \end{aligned} \quad (4)$$

$$\begin{aligned} \mu_{<0>} &= E\{\min(T_0, T_1, ..., T_{P-1})\} \\ &= \int_{-\infty}^{\infty} [S(x) - (1 - \prod_{i=0}^{P-1} (1 - F_{T_i}(x)))] dx \end{aligned} \quad (5)$$

where $F_{T_i}(x)$ denotes the cumulative function of random variable T_i, and

$$S(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Since arithmetic mean of a set of non-negative numbers is no less than the corresponding geometric mean, it follows that

$$(\frac{\sum_{i=0}^{P-1} F_{T_i}(x)}{P})^P \geq \prod_{i=0}^{P-1} F_{T_i}(x)$$

Equation (4) then becomes

$$\mu_{<P-1>} \geq \int_{-\infty}^{\infty} [S(x) - (\frac{\sum_{i=0}^{P-1} F_{T_i}(x)}{P})^P]dx \quad (6)$$

By further applying the following generalized Cauchy-Schwartz inequality

$$\sum_{i=0}^{P-1} \frac{F_{T_i}^P(x)}{P} \geq (\frac{\sum_{i=0}^{P-1} F_{T_i}(x)}{P})^P$$

it becomes

$$\begin{array}{rcl} \mu_{<P-1>} \geq & \int_{-\infty}^{\infty} [S(x) - \sum_{i=0}^{P-1} \frac{F_{T_i}^P(x)}{P}]dx \\ = & \frac{1}{P}\sum_{i=0}^{P-1} \int_{-\infty}^{\infty} [S(x) - F_{T_i}^P(x)]dx \end{array} \quad (7)$$

Let T_i, $T_{i\{1\}}$, $T_{i\{2\}}$, $\cdots$, $T_{i\{P-1\}}$ be a set of P identical independent distributed (i.i.d.) random variables, where $0 \leq i \leq P-1$. We have

$$\int_{-\infty}^{\infty} [S(x) - F_{T_i}^P(x)]dx = E\{\max(T_i, T_{i\{1\}}, \cdots, T_{i\{P-1\}})\}$$

With a result in [10]

$$\int_{-\infty}^{\infty} [S(x) - F_{T_i}^P(x)]dx = \mu_i + \sigma g(P) \quad (8)$$

where

$$g(P) = \int_{-\infty}^{\infty} [\int_{-\infty}^{u} f_{T_i}(v)dv - (\int_{-\infty}^{u} f_{T_i}(v)dv)^P]du$$

and f_{T_i} is the *pdf* of random variable T_i. Note that $g(P)$ can be derived with various numerical methods or by approximation [10]. From inequality relation (7) and Equation (8),

$$\begin{array}{rcl} \mu_{<P-1>} \geq & \frac{1}{P}\sum_{i=0}^{P-1}(\mu_i + \sigma g(P)) \\ = & \frac{1}{P}\sum_{i=0}^{P-1} \mu_i + \sigma g(P) \\ = & \mu_0 + \sigma g(P) + \frac{P-1}{2}\delta_\mu \end{array} \quad (9)$$

On the other hand, it is obvious that

$$\begin{array}{rcl} \mu_{<P-1>} = & E\{\max(T_0, T_1, ..., T_{P-1})\} \\ \leq & E\{\max(T_{P-1}, T_{P-1\{1\}}, \cdots, T_{P-1\{P-1\}}\} \\ = & \int_{-\infty}^{\infty} [S(x) - F_{T_{P-1}}^P(x)]dx \\ = & \mu_{P-1} + \sigma g(P) \\ = & \mu_0 + \sigma g(P) + (P-1)\delta_\mu \end{array} \quad (10)$$

where $T_{P-1}, T_{P-1\{1\}}, \cdots, T_{P-1\{P-1\}}$ are again i.i.d. random variables as defined. Combining the two inequality relations (9) and (10), we have

$$\mu_0 + \sigma g(P) + \frac{P-1}{2}\delta_\mu \leq \mu_{<P-1>} \leq \mu_0 + \sigma g(P) + (P-1)\delta_\mu \quad (11)$$

With a similar procedure applied for $\mu_{<0>}$, we have

$$\begin{array}{rcl} \mu_{<0>} = & E\{\min(T_0, T_1, ..., T_{P-1})\} \\ = & \int_{-\infty}^{\infty} [S(x) - (1 - \prod_{i=0}^{P-1}(1 - F_{T_i}(x)))]dx \\ = & \int_{-\infty}^{\infty} [S(x) - 1 + \prod_{i=0}^{P-1}(1 - F_{T_i}(x))]dx \\ \leq & \int_{-\infty}^{\infty} [S(x) - 1 + (\frac{\sum_{i=0}^{P-1}(1-F_{T_i}(x))}{P})^P]dx \\ \leq & \int_{-\infty}^{\infty} [S(x) - 1 + \sum_{i=0}^{P-1} \frac{(1-F_{T_i}(x))^P}{P}]dx \\ = & \frac{1}{P}\sum_{i=0}^{P-1} \int_{-\infty}^{\infty} [S(x) - (1 - F_{T_i}(x))^P]dx \end{array}$$

Again with

$$\int_{-\infty}^{\infty} [S(x) - (1 - F_{T_i}(x))^P]dx = E\{\min(T_i, T_{i\{1\}}, ..., T_{i\{P-1\}})\}$$

and an extended result from [10], it follows that

$$\int_{-\infty}^{\infty} [S(x) - (1 - F_{T_i}(x))^P]dx = \mu_i - \sigma g(P)$$

Thus,

$$\begin{array}{rcl} \mu_{<0>} \leq & \frac{1}{P}\sum_{i=0}^{P-1}(\mu_i - \sigma g(P)) \\ = & \mu_0 - \sigma g(P) + \frac{P-1}{2}\delta_\mu \end{array}$$

Combined with the following inequality

$$\begin{array}{rcl} \mu_{<0>} = & E\{\min(T_0, T_1, ..., T_{P-1})\} \\ \geq & E\{\min(T_{0\{0\}}, T_{0\{1\}}, \cdots, T_{0\{P-1\}}) \\ = & \mu_0 - \sigma g(P) \end{array}$$

a bound of $\mu_{<0>}$ is then derived as

$$\mu_0 - \sigma g(P) \leq \mu_{<0>} \leq \mu_0 - \sigma g(P) + \frac{P-1}{2}\delta_\mu \quad (12)$$

Further combining the two inequality relations (11) and (12), we then have

$$2\sigma g(P) \leq \mu_{<P-1>} - \mu_{<0>} \leq 2\sigma g(P) + (P-1)\delta_\mu \quad (13)$$

Substituting $\mu_{<P-1>} - \mu_{<0>}$ with the relation in Equation (3) in Equation (13), we obtain

$$2\sigma g(P) \leq (P-1)T_{comm} \leq 2\sigma g(P) + (P-1)\delta_\mu$$

or simply

$$0 \leq T_{comm} - \frac{2\sigma}{P-1}g(P) \leq \delta_\mu \quad (14)$$

The inequality relation on the right gives a lower bound for δ_μ when selecting such a skewing mass in an *SLA* approach. This bound is tight if the skewing amount is not large. That is, if the skewing amount needed is expected to be relatively smaller than μ_i's, δ_μ can be selected to be very close to

$$T_{comm} - \frac{2\sigma}{P-1}g(P)$$

On the other hand, the left inequality relation indicates that in order for any *SLA* approach to benefit the overall performance, σ cannot exceed a certain threshold, which is represented by

$$\sigma \leq \frac{(P-1)T_{comm}}{2g(P)} \quad (15)$$

That is, a simple *PBLA* would be sufficient if σ exceeds this threshold.

3 Simulation and Experiment

3.1 Simulation

In our simulation process to confirm the proposed analytical results, the following parameters are given as inputs: 1) mean of total computation load ($\mu_{T_{comp}}$), 2) number of workstations (P), 3) required communication time for each workstation (T_{comm}), and 4) standard deviation of each subtask's load performed in a workstation (σ). Data for random variables T_0, T_1, ..., T_{P-1} are randomly generated according to their corresponding normal *pdf*s, $N(\mu_0, \sigma)$, $N(\mu_1, \sigma)$, ..., $N(\mu_{P-1}, \sigma)$, respectively. By varying δ_μ (and thus all μ_i's), overall performance is plotted in Figure 3 (for $P = 2$) and in Figure 4 (for $P = 4$) under different σ values. From the left figures, when $\sigma = 0$, the over-

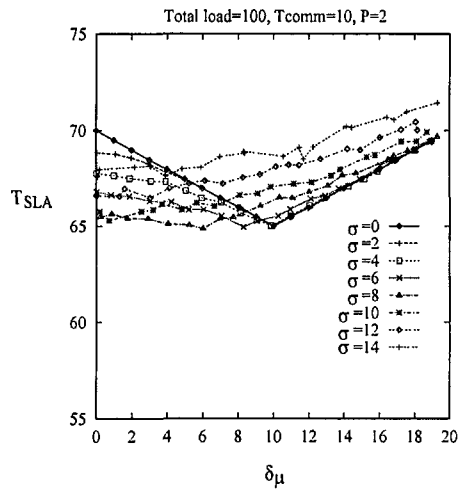


Figure 3: Simulation Results ($P = 2$)

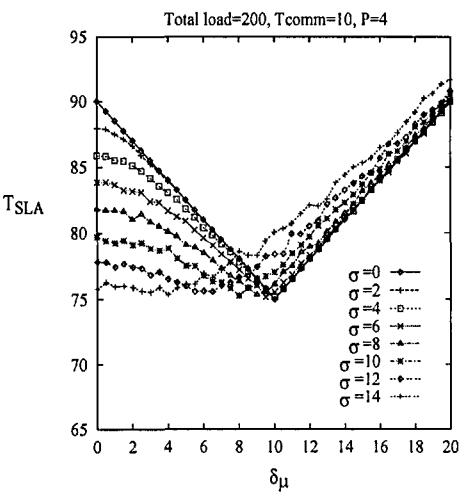


Figure 4: Simulation Results ($P = 4$)

all time is minimized by having $\delta_\mu = T_{comm}(= 10)$ just like a deterministic case. As σ increases, the optimal skewing mass δ_μ decreases as expected. From the right figures, we can tell that picking the optimal skewing mass always leads to the best effective skewing mass T_{comm} when σ does not exceed a threshold. The optimal skewing mass observed in our simulation is then plotted against the corresponding σ values and is compared with the analytical bound in Equation (14). This is shown in Figure 5. From these, we can conclude

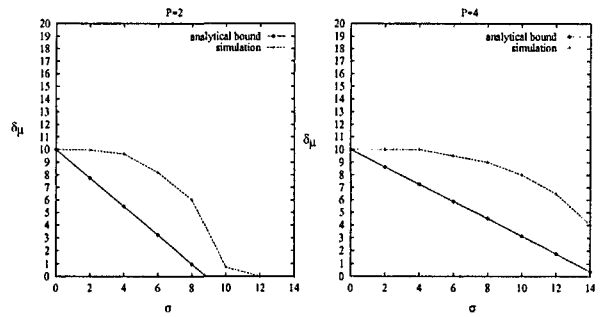


Figure 5: Comparison between Analytical and Simulation Results

that the analytical bound is close to the optimal skewing mass we need to minimize communication conflict. Another critical observation is that when σ exceeds a threshold (about 10 for $P = 2$ and 15 for $P = 4$), no more skewing is needed. This also closely matches our analytical result in Equation (15) which predicts, for $P = 2$,

$$\sigma \leq \frac{10}{2 \times 0.564190} \approx 8.86$$

and, for $P = 4$,

$$\sigma \leq \frac{3 \times 10}{2 \times 1.029376} \approx 14.57$$

3.2 Simulated Experiment

We further carry out a simulated experiment with a PVM (Parallel Virtual Machine) environment on a real bus-based *NOW*. A master/slave parallel programming configuration is used in which a master processor collects messages from slave processors at the end of each slave's computation process. In this experiment, computing time required in each slave processor is generated by a random number generator obeying the aforementioned normal distribution with various σ values. A simulated communication stage is provided by having each slave processor send a package of 5,000 double precision floating point numbers to the master processor. Such a communication step is measured to take an average time of $T_{comm} = 0.11$ seconds. Figure 6 and Figure 7 give the results from this experiment for $P = 2$ and $P = 4$ respectively. Optimal skewing masses observed are further compared with the derived analytical bound. Figure 8 displays the comparison result. Again the results match very nicely.

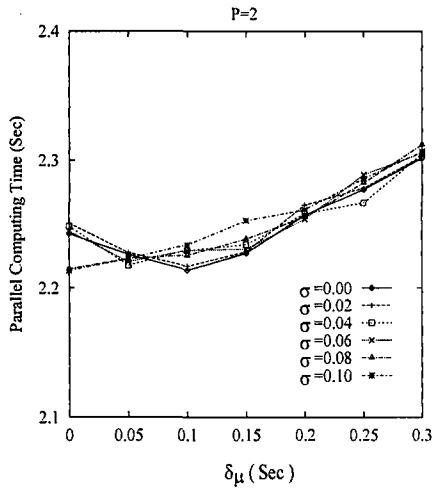


Figure 6: Simulated Experimental Results ($P = 2$)

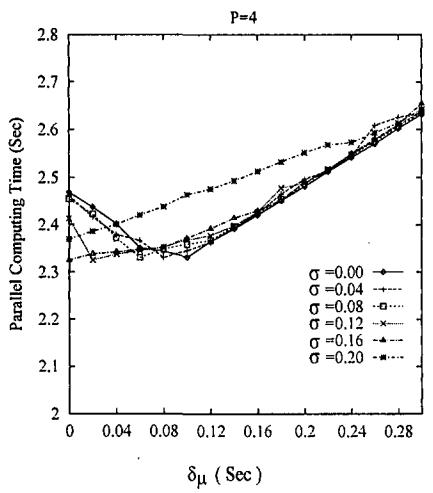


Figure 7: Simulated Experimental Results ($P = 4$)

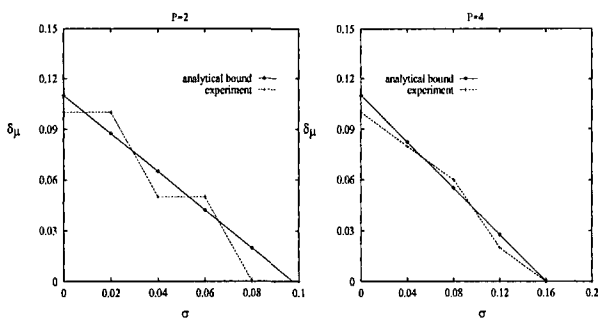


Figure 8: Comparison between Analytical and Experimental Results

4 Real Examples

To further verify our analysis, we use two widely used programming examples, a parallel matrix-vector multiplication algorithm and a one-stage parallel sorting algorithm. Disregarding nondeterministic factors caused by the machines, the matrix-vector multiplication example is used to verify the deterministic case, while the sorting algorithm which is associated with a nondeterministic nature in its computation requirement will be used for the nondeterministic case.

4.1 Matrix-Vector Multiplication

In this matrix-vector multiplication example, a multiplication of an $n \times n$ matrix A and an n-component vector V is performed with a PVM environment on a bus-based P-workstation network. The matrix A is partitioned into P disjoint submatrices A_i, $0 \leq i \leq P-1$, each containing n_i continuous rows in matrix A. Thus, $\sum_{i=0}^{P-1} n_i = n$. Each processor will calculate the corresponding n_i components of the result vector before sending its results to a master processor. A simple loop construct is used to calculate each component in the result vector, which requires a deterministic amount of computation time, the time for n multiplication steps and $n-1$ addition steps to be precise.

In this example, since the computation time required in a processor is linearly proportional to the number of rows in A (n_i) assigned to it, the skewing mass δ_μ can be adjusted accordingly by varying the number of rows assigned to different processors. Experiments are performed on the cases of $n = 512$ and $n = 1024$. Figure 9 shows the performance results versus the varying parameter δ_μ. The computing time

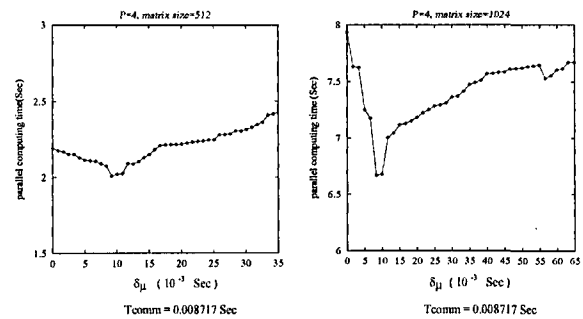


Figure 9: Performance Results of Matrix-Vector Multiplication

difference in a processor caused by assigning one extra row of A to it is measured at about 0.418msec. The required communication time for each slave processor is measured at $T_{comm} = 8.1717$msec. When the skewing mass δ_μ is set at 8.36msec, i.e. by having $n_{i+1} - n_i = 20$ ($20 \times 0.418 = 8.36$), close to optimal performance is observed. This skewing mass is

very close to T_{comm} (since $\sigma = 0$), the theoretical optimal value. Similar prediction accuracy is observed in the $n = 1024$ case, where load skewing is set by $n_{i+1} - n_i = 10$. This leads to a speedup improvement of 10.2% and 19.6% on the two test cases respectively using the proposed *SLA* approach over the simple *PBLA* one.

4.2 Parallel Sorting

In this example, 10,000 double-precision floating point numbers are to be sorted with a network of P workstations. The master processor first partitions the numbers into P disjoint sets and then allocates each to a distinct slave processor. Each slave processor then performs a sequential quick sort process on its assigned data set before sending the sorted sequence back to the master processor. A merge process is conducted on the master processor to reach the final sorted sequence. Note that the goal of this example is not to evaluate efficiency of a parallel algorithm, but to verify our analysis finding. With such a solution algorithm, the exact computation time on each slave processor (T_i) is a nondeterministic amount due to its run-time data dependency.

In this experiment, a random number generator is used to generate the target number sequences. By using a different modulus (which is also the range of the generated numbers), we are able to create sequences which would lead to different σ values. This is needed in order to measure the effects of σ on the optimal skewing mass. Four different moduli are used, 1, 10, 100 and 1000 respectively, to generate four number sequences to be tested. Two network configurations are used, one with $P = 4$ and the other with $P = 8$. Communication time needed for each slave processor to send its results back to the master, T_{comm}, is measured at 0.064926sec for $P = 4$ and 0.050338sec for $P = 8$ respectively. Experimental results are shown in Figure 10. Note that the skewing mass δ_μ changes

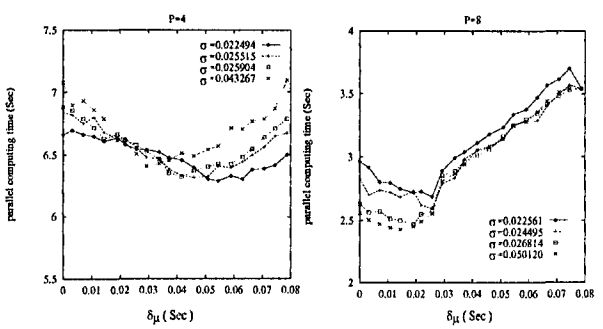


Figure 10: Performance Results of Parallel Sorting

monotonically to the change in the sizes of data set assigned to the processors, although such a change is in a non-linear fashion. With optimal δ_μ selected, speedup improvement in using the *SLA* approach versus the simple *PBLA* ranges from 10% to 14% on the two network configurations.

Experimental results on optimal skewing mass compared with the derived analytical bound are displayed in Figure 11. In the $P = 4$ case, T_{comm} is measured at

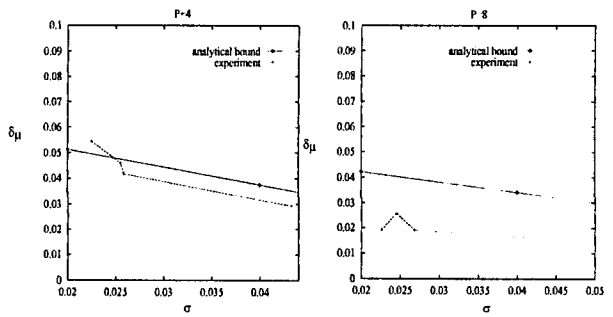


Figure 11: Comparison between Analytical and Real Experiment Results

0.064926sec. Thus, the maximal σ that would still call for an *SLA* approach to reduce communication conflict is

$$\sigma \leq \frac{3 \times 0.064926}{2 \times g(4)} = 0.09461$$

which is not within the scope of our experiment, as shown in the comparison figure. In the $P = 8$ case, T_{comm} is measured at 0.050338sec. Similarly, the maximal σ that would still require an *SLA* approach is

$$\sigma \leq \frac{7 \times 0.050338}{2 \times 1.423602} = 0.123759$$

which also falls outside of our experiment scope. However, the trend of the two does indicate an excellent match. On optimal skewing masses, the discrepancies between the analytical bound and the experimental values could be a result of several factors. The fact that a non-dedicated *NOW* system, rather than a dedicated one as required in our analysis, is used for the testbed could be a leading contributing factor. Ignoring extra delay caused by possible collisions on the bus in our analysis can explain why an effective skewing mass smaller than the analytical lower bound is sufficient in leading to optimal overall performance.

5 Conclusion

In this paper, we have provided a thorough analysis on how communication conflict can be minimized in a bus-based network of workstations by using a load-skewing assignment method. The analytical model is validated by extensive simulation and experimental results. Significant speedup improvement was also

shown by using such a new method. Moreover, we believe that this model can be furthered extended to analyze more complicated solution algorithm constructs. A multiple-stage construct has been under investigation with very promising preliminary results.

Acknowledgements

This research was supported in part by the Office of Naval Research under grant N00014-95-1-0514 and N00014-96-1-0897, and in part by the Department of Defense/Air Force Office of Scientific Research under grant F49620-96-1-0472.

References

[1] V. S. Adve and M. K. Memon, "The Influence of Random Delays on Parallel Execution Times," *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.

[2] M. Andrews, T. Leighton, P. T. Metaxas, and L. Zhang, "Improved Methods for Hiding Latency in High Bandwidth Networks." *SPAA'96.* pp.52-61, Padua, Italy. 1996.

[3] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computation by Work Stealing," *Proc. of Ann. Symp. on Foundations of Computer Science.* pp.356-368, Nov. 1994.

[4] T. Casavant and J. Kuh, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. Software Eng.* SE-14(2), pp.141-154, Feb. 1988.

[5] H. A. David, *Order Statistics.* New York: Wiley, 1981.

[6] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Software Eng.* SE-12(5), pp.662-675, May 1986.

[7] S. Flynn Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," *Comm. of the ACM.* 35(8). pp.90-101, August 1992.

[8] H. O. Hartley and H. A. David, "Universal Bounds for Mean Range and Extrema Observations". *Ann. Math. Statist.* Vol.25, pp. 85-99, 1954.

[9] C. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors." *IEEE Trans. Software Eng.* SE-11(10), pp. 1001-1016, Oct. 1985.

[10] W.-M. Lin and B. Yang, "Load Balancing Technique for Parallel Search with Statistical Model," *1995 International Phoenix Conference on Computers and Communications.*

[11] W.-M. Lin and Z. Yun, "Performance Analysis on Divide-and-Conquer Parallel Search Techniques," *International Conference on Parallel and Distributed Systems*, 1993

[12] D. C. Marinescu and J. R. Rice, "Synchronization and Load Imbalance Effects in Distributed Memory Mult-processor Systems", *Concurrency: Practice and Experience*, Vol. 3, pp593-625, Dec. 1991.

[13] S. Madala and J. B. Sinclair, "Performance of Synchronous Parallel Algorithms with Regular Structures," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2, No. 1, Jan. 1991.

[14] H. Nishikawa and P. Steenkiste, "A General Architecture for Load Balancing in a Distributed-Memory Environment," *Proc. 19th Intl. Conf. on Distributed Computing.* pp.47-54, May 1993.

[15] V. N. Rao and V. Kumar, "On the Efficiency of Parallel Backtracking", *IEEE trans. on Parallel and Distributed Systems*, Vol.4, No.4, April 1993.

[16] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, "A Simple Load Balancing Scheme for Task Allocation in Parallel Machines," *Proc. of Sym. on Parallel Algorithms and Architectures.* pp.237-245, 1991.

[17] M. Schmidt-Voigt, "Efficient Parallel Communication with the nCUBE 2S Processor." *Parallel Computing.* Vol.20, No.4, April 1994.

[18] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *IEEE Computer.* pp.33-45, Dec. 1992.

[19] A. B. Tayyab and J. G. Kuhl, "Stochastic Performance Models of Parallel Task Systems", *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp284-285, May 1994.

[20] L. W. Tucker and A. Mainwaring, "CMMD: Active Messages on the CM-5." *Parallel Computing.* Vol.20, No.4, April 1994.

[21] Y. T. Wang and R. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. Computers.* C-34(3), pp.204-217, Mar. 1985.

[22] L. F. Wilson and M. J. Gonzalez, "Synchronization and Communication in Algorithmic Structures", *Proc. of the 6th IEEE Symp. on Paral. and Distri. Processing*, pp196-203, 1994.

[23] W. Xie and W.-M. Lin, "Communication Conflict Avoidance on Bus-Based Networks of Workstations with Load-Skewing Task Assignment Techniques," *Technical Report*, Div. of Engineering, UT-San Antonio, May 1997.

Scheduling of I/O in Multiprogrammed Parallel Systems

Peter Kwong and Shikharesh Majumdar
Department of Systems and Computer Engineering
Carleton University, Ottawa, CANADA K1S 5B6
Phone: +1 613 520 5654, +1 613 520 5727
E-mail: majumdar@sce.carleton.ca

Keywords: multiprogrammed parallel systems, parallel I/O, scheduling, performance evaluation

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 25, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

Recent studies have demonstrated that significant I/O is performed by a number of parallel applications. In addition to running these applications on multiple processors, the parallelization of I/O operations and the use of multiple disk drives are required for achieving high system performance. This research is concerned with the effective management of parallel I/O by using appropriate I/O scheduling strategies. Based on a simulation model the performance of a number of scheduling policies are investigated. Using I/O characteristics of jobs such as the total outstanding I/O demand is observed to be useful in devising effective scheduling strategies.

1 Introduction

The continuous growth in CPU and memory speeds, as well as the deployment of parallel processing technology, have significantly reduced the computation times for applications. Because of their electro-mechanical construction, however, the access time for disks has improved only minimally over the past twenty years [20]. As a result, in many situations, the performance bottleneck has shifted from the processor subsystem to the I/O subsystem. Significant I/O is performed in different classes of parallel applications that include the grand challenge programs and scientific applications [20] as well as graphics software [5]. To reduce the intrinsic problems of the slow electro-mechanical technology used to build I/O devices, disk caches and arrays of disks have been introduced (see [19] for example). Another approach to improve system performance is to use parallel I/O. Although multiple I/O devices can potentially improve system performance, parallelization of I/O in an application as well as appropriate management of the I/O devices are required to harness the power of the parallel I/O subsystem. This paper is motivated by such requirements on the effective management of parallel I/O in general and scheduling of parallel I/O in particular.

A number of commercial shared memory as well as distributed memory multiple processor systems are currently in use. Examples include symmetric shared memory multiprocessor (SMP) systems produced by Sequent and Encore; processes or threads in an application exchange information through shared variables stored in the global memory provided on such a system. Distributed memory systems on which processes communicate through message passing include the Intel's Hypercube and the NCube systems. Non-uniform-memory-access (NUMA) systems such as the Teracomputer, and the KSR-1 are hybrid between the shared and distributed memory classes. The availability of processors as well as high speed inter-connection networks at a reasonable cost has created a new trend called cluster-based computing the popularity of which is rising rapidly. A cluster is a group of inter-connected computers working as a unified computing resource [24]. A network of multiprocessor workstations is an example of a cluster system. In addition to its attractive price-performance ratio, a cluster also provides both absolute as well as incremental scalability and high availability.

The availability of workstation clusters and other parallel hardware, along with tools such as restructuring compilers for developing parallel application software, are increasing the popularity of parallel systems. A number of existing systems are dedicated to running a single computation and I/O intensive application. However as the usage of multiprocessor and cluster systems are becoming more and more widespread, general purpose systems that run a number of different parallel applications are becoming popular. Environments running such a variety of different applications need multiprogramming to provide user satisfaction and enhance resource utilization. Multiple parallel applications are active simultaneously on such an environment.

Although multiprogramming provides user satisfaction and improves resource utilization, a scheduler that mediates among the demands of competing jobs is required for each resource. Existing research on resource

management in multiprogrammed parallel systems has focused on how to schedule processors among applications in the multiprogramming mix to achieve a small mean job response time (see for example [15, 16, 23]). Job scheduling on a network of workstations has recently started receiving attention [1] A significant amount of research exists on processor scheduling in both shared and distributed memory systems but is not discussed here in detail due to limitations of space.

Existing work on parallel I/O has focused primarily on environments that run a single application in isolation. A number of I/O subsystem architectures is discussed in the context of parallel systems in [7], whereas a comparison of different disk configurations that include disk striping and disk synchronization for a transaction system workload as well as for a scientific application workload, is presented in [21]. The performance of two RAID architectures and a number of scheduling policies is considered in [2] whereas performance modelling of disk drives and data caching are considered in [22]. Characterization of file access patterns exhibited by parallel scientific workload is discussed in [10]. Scheduling I/O requests for reducing the total completion time for the schedule is discussed in [8] whereas techniques for a co-ordinated management of processing and I/O resources in a multiprogrammed parallel system are presented in [23]. Research on parallel file systems is also underway. The Galley file system that is composed of a standard fixed core and a set of platform dependent libraries is described in [11]. The Hurricane file system that provides a collection of building block objects that can be plugged together differently for different system needs is discussed in [14]. User defined file partitioning and dynamic decomposition of files in a parallel file system is considered in [4].

None of the existing studies described in the previous paragraph has addressed a number of basic issues underlying the performance of parallel I/O on a *multiprogrammed* parallel system. The style in which an application performs I/O, the strategies for scheduling parallel I/O, and data distribution on multiple disks can have significant impacts on the performance of a multiprogrammed system. Some work on characterization of I/O and techniques for distribution of data on multiple disks has been reported in [18] and [12] respectively. The viability of parallel I/O on a network of workstations and the resulting performance improvement are described in [3] and [6]. This paper presents some new results that focus on the scheduling of I/O requests in a multiprogrammed parallel system. A number of scheduling policies are proposed and their performances under different data distribution strategies and I/O styles used by applications are discussed.

Based on an abstract simulation model of systems, applications, data distribution, and scheduling policies, a number of high level questions that are important for the operating system on a cluster running a multiprogrammed parallel workload, as well as on multiprogrammed shared memory environments are discussed. The issues addressed by this research include the following.

- Using characteristics of applications have been found to be beneficial in CPU scheduling on both single as well as multiprocessor systems. Are job characteristics important in the context of scheduling I/O? How do the different attributes of parallel applications affect the relative performances of the I/O scheduling policies?
- Sharing the CPU equally among a number of competing applications is found to be effective on conventional multiprogrammed systems. How useful is this principle of equal sharing in the context of scheduling parallel I/O?
- Different ways in which an application performs I/O, as well as strategies for replicating and distributing data over multiple disks, are found to have a significant effect on system performance [12]. How do these interact with the different scheduling policies to determine the overall system performance?
- Management of I/O may be performed in a number of different ways. A centralized management of I/O requests may be appropriate for small to medium scale parallel systems, whereas a decentralized approach is necessary for larger systems. How do the distributed and centralized approaches to I/O management affect the relative performance of the scheduling policies?

These high level questions are investigated in the context of multiprogrammed systems that consist of a number of independent processing and disk resources. Such a loosely coupled shared every thing environment [7] is provided for example by "shared disk" clusters described in [24]. Each set of processing resources constitutes a compute server or node. These compute servers share a common set of I/O servers or nodes. This paper focuses on systems that run applications characterized by large compute and I/O transfer times. The device seek times are assumed to be much smaller in comparison.

Computer simulations are run to investigate the questions outlined above. The simulation results are calculated to a 95% confidence level with an interval which is less than $\pm$ 5% in most cases. The paper is organized as follows. The simulation model is introduced in the following section. Descriptions of the data distribution strategies and of the scheduling policies are presented in Section 3 and Section 4 respectively. The simulation results are described in the following section. Section 6 presents our conclusions.

2 The System Model

The simulation model for an open system in which jobs arrive from the external environment consists of two components: the model of the multiprogrammed system and that of the application workload. The multiprogrammed system model is presented first and the workload model is described in the following subsection.

2.1 Multiprogrammed System Model

The multiprogrammed system consists of the compute and the I/O subsystems. Statistically identical jobs run on the system. During its execution, a job does computation, requests I/O service from the I/O subsystem, and exits the system when all its operations are complete. The compute and I/O subsystems are characterized by the number of processors, P, and the number of independent I/O nodes, I, respectively. One or more of these I/O nodes can be accessed in parallel by the threads of the same application running on different processors. The parallelism in I/O access is discussed further in the following subsection.

Jobs arrive as a Poisson process characterized by an arrival rate λ into the compute subsystem. The P identical CPUs in the compute subsystem are grouped into M processor sets (job nodes). Each set has Np CPUs, but if P is not a multiple of Np, then one of the sets will have only P-Np*(M-1) CPUs. On a cluster of workstations for example, each set corresponds to a compute node consisting of Np processors. On a shared memory multiprocessor this corresponds to a static space sharing system consisting of P processors in which the degree of multiprogramming is upper bounded by M. Each job that enters the system is sent to the first free job node. If no job node is idle, then the newly arrived job waits in a FIFO queue for the first idle job node. Static scheduling of processors is used, and once a job acquires a job node all the processors are held by the job until job completion. Such a CPU scheduling strategy has been investigated by a number of researchers (see [24, 16] for example) and is found to be effective in a number of different environments. Processors allocated to the same job are sharable by the threads in the job that share the same address space.

Following an I/O request, a thread in a job is blocked waiting for the I/O request to finish. During this period, the CPU on which the thread was running becomes available for use by other threads in the same job. Processors may be shared by threads in an application in different ways. In this research a First-Come-First-Served policy is used and a ready thread is assumed to run on the first available processor. An I/O request from a job is sent to one of the I/O nodes in the I/O subsystem. Each I/O node operates independently, has the same speed, and is characterized by a mean seek time ds, as well as a coefficient of variation of seek time, CV_s. Note that although each I/O node may consist of an array of disks it is modelled as a single I/O node with a given set of seek time parameters. The data transfer time from the disk (also called the I/O demand in this paper) for an I/O request is discussed further in the following subsections.

This research is concerned with the performance of scheduling strategies characterized by large computational workloads and I/O transfer times. In all the simulation experiments described in this paper the mean seek time is assumed to be small in comparison to data transfer time and is held at 1% of the mean thread execution time. CV_s is fixed at 1.0. For simplicity, overheads associated with CPU and I/O scheduling as well as context switching are ignored. This is appropriate for the qualitative nature of the questions investigated. We have also ignored the contention for the interconnection network that connects the I/O nodes with the job nodes. This is appropriate when the speed of the network is high such that the network delay is small compared to the disk transfer times. Moreover this paper is concerned with gaining insights into I/O scheduling under a number of different data replication strategies and application classes. Isolating the network delay from the performance analysis is also necessary for obtaining a clear understanding of I/O scheduling which is the focus of this paper. The performance measure of interest is the mean job response time, R. The response time of a job is the difference between the time at which the last thread in the job completes and the arrival time for the job. The mean response time R is normalized with respect to the mean thread computation time.

2.2 Workload models with Parallel I/O

Jobs with a fork and join architecture, used in existing work on processor management on parallel systems (see [15] for example), are adapted by this research to represent I/O performed by jobs. Research on processor scheduling in parallel systems has ignored the I/O performed by jobs. This fork-and-join model is augmented to incorporate different styles in which applications may perform I/O. Each of this I/O style gives rise to a different job model. The workload applied to the system in the simulation experiments is either the O1 (Overlap # 1) or the NOP (No Overlap-Parallel) job models (see Figure 1) introduced in [17] and [13]. The NOP job is characterized by no CPU-I/O overlap and the O1 job is characterized by CPU-I/O overlap. Both jobs characterized by CPU and I/O overlap as well as no CPU and I/O overlap have been observed in scientific programs and graphics software [5]. Similar observations for parallel scientific

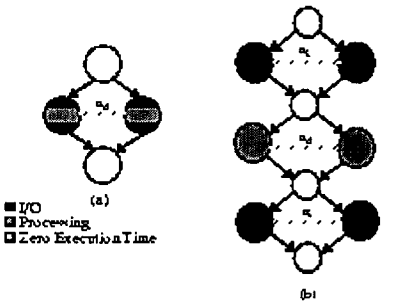


Figure 1: The Job Models. (a) O1 Model (b) NOP Model

workload have been reported by other researchers as well. These two models are described briefly.

The O1 Model: This represents jobs which overlap I/O with computations. A job starts by forking n_d children threads each of which reads data from an I/O node then executes on the CPU. After completing its computation, each thread writes data onto an I/O node. When all the children threads have completed their writes, the job terminates (see Figure 1(a)).

The NOP Model: A NOP job begins by first reading in data. The read, however, is done using n_i parallel I/O operations from more than one I/O node. When all the read operations have completed, the job forks n_d children threads, each of which only performs computations. When all these threads complete their computations, the job performs n_i parallel write operations to more than one I/O node. After all the writes are completed, the job terminates (see Figure 1(b)).

Two other I/O models were also investigated (see [13]) and are not reported here for the conservation of space.

The job models can be characterized by using a common set of parameters as well as parameters specific to a particular I/O model. An important characteristic of parallel applications, called the I/O factor, was introduced in [17]. The I/O factor is the ratio between the mean total I/O demand (IOD) and the mean total CPU demand of the job: $Fi = IOD/(n_d * d)$ where d is the mean CPU demand for a thread. The I/O factor reflects the relative weight of the computation and I/O workload components of the applications. All computation and I/O demand related parameters are normalized with respect to the mean thread execution time d.

The random variable for the CPU burst duration of a thread is generated by using d as the mean and CV_d as the coefficient of variation. For a given F_i and d, the random variable for an I/O burst duration is generated by using the I/O request demand ($IOD/2n_d$ for O1 and $IOD/2n_i$ for NOP) as the mean and CV_i as the coefficient of variation. Generation of these random variables is based on a bi-phase hyper-exponential distribution ([9]).

3 Data Distribution Policies

Previous studies have shown that the way data read or written by applications are distributed on the available disks can have a significant impact on performance. I/O scheduling under two classes of data replication strategies are investigated: Write Anywhere Read Anywhere ("Replicated"), and Fixed Read and Write ("Fixed Read Write"). The Replicated policies are further subdivided into Centralized (Replicated-C) and Distributed (Replicated-D). In the centralized approach, a centralized dispatcher monitors the status of all I/O nodes (busy or idle) and requests are always sent to an idle node. In the distributed approach, an automatic routing of an I/O request to a node is made without regard or knowledge of the current node status (busy or idle). The centralized approach is appropriate for smaller parallel systems. In larger systems, there may be a high cost associated with maintaining such a centralized database and a distributed approach for I/O management may be preferable.

The Replicated policies use replication of data read by applications whereas only a single copy of input data is maintained by the Fixed Read Write policy. A program's input data is assumed to be available on all the I/O devices for the replicated policies, whereas the input data is available on only one disk for the Fixed Read Write policy. For all the policies only a single copy of data is written, however, by the applications. Thus data written by a thread can be sent to any one I/O node irrespective of where the data written by other threads in the application are sent. Note that the write operations performed by the applications are assumed to correspond to the writing of the output results only and the written data is not read by any thread in the application. This is appropriate for a variety of different types of applications such as scientific programs that this research is concerned with. Replication of written data may be required in other types of environments, such as transaction processing systems. With the replicated data distribution strategies any I/O node can be used to service any I/O

request. For FixedRW, no data replication is used. A read or write operation is directed to a specific I/O node.

In Replicated-C, all I/O requests are inserted into a central queue ordered in accordance with the I/O scheduling policy in use. A central router which knows the busy/idle status of every I/O node initiates a transfer between the requesting thread and the first idle I/O node the router finds. If no node is idle, then all requests remain in the queue until a node becomes free. In the Replicated-D policy, each request is sent to one of the I nodes chosen at random with probability 1/I. At the node, requests are then placed into a queue maintained and accessed only by the local I/O node. The organization of the queue depends on the scheduling policy that is described in the following section. If the node is idle, then the request is processed immediately.

Most existing parallel systems use the FixedRW policy that does not use any data replication. Data can be written by an application only to specific nodes and data can be read only from specific nodes. For FixedRW, the data read by the threads in an application is distributed in a Round Robin fashion. With the Round Robin allocation, parallelism in I/O is maximized and $min(I, n_i)$ nodes are used by an NOP application, whereas $min(I, n_d)$ nodes are used by an O1 application. Data for the first thread is assumed to be placed in a node k chosen at random by the simulator. The data for the next thread is assumed to be placed on $k+1$ and so on. The data for all the threads are placed on the nodes in such a round robin fashion. When all the nodes are used once, node k and other nodes may be used again. The performances of these data distribution strategies were investigated in isolation of scheduling in [12]. Replicated-C was observed to produce the best performance. However, if the cost of maintaining a centralized database of node status is too high on a system, a Replicated-D or FixedRW strategy needs to be adopted. The relative performance of Replicated-D and FixedRW was observed to depend on a number of different workload and system parameters. In most situations FixedRW demonstrated a better performance if Round Robin data distribution was used whereas Replicated-D seems to be preferable when a Round Robin data organization is too expensive on the multiprogrammed system and a simpler variant is used with FixedRW. A detailed discussion of the different versions of FixedRW is presented in [13]. Two other variants of the replicated policies were also investigated by the authors and their performances were observed to lie in between Replicated-C and Replicated-D [13]. In order to keep the number of experiments to a reasonable number, we have decided to investigate the scheduling strategies presented in the following section under three different data distribution strategies: Replicated-C, Replicated-D, and FixedRW (Round Robin). The first is a representation of the best performance that can be expected in systems with centralized control. The second is appropriate if a centralized control is not possible, whereas the third may have to be used if the cost of data replication is too high for the user. In all further discussion FixedRW will imply FixedRW (Round Robin).

4 I/O Scheduling Policies

Existing disk scheduling algorithms such as SSTF, SCAN, LOOK etc [24] are used on conventional systems to select one I/O request from a list of several outstanding requests at a single I/O node. Because these algorithms attempt to minimize the overhead (seek times) associated with a disk access by reducing disk head movements, they are effective where disk seek times are significant relative to the data transfer times.

The I/O scheduling policies proposed in this paper are intended for scheduling requests on systems in which the data transfer time is much greater than the access overhead. These policies use job characteristics such as the age of the jobs ("JBOJF") or the total outstanding I/O demand of the jobs ("JBSOIO" and "JBLOIO") for determining the priority of an I/O request. For investigating the importance of equal sharing a round robin scheduling policy ("JBRR") is introduced. The performances of these four policies are compared with the performance of a scheduling policy that does not use any knowledge of job attributes while serving a request from the job ("JBNOT"). For the JBNOT policy, I/O requests are processed on a First-Come-First-Served (FCFS) basis.

I/O scheduling can be done either by a central scheduler, or by a scheduler local to each I/O node. Note that depending on the data distribution strategy in use, any of these scheduling policies may be used to manage a global request queue or an individual local queue at each I/O node. A central scheduler is appropriate if a centralized data replication policy such as Replicated-C which performs centralized routing of I/O requests is used. A local scheduler is appropriate if the Replicated-D or FixedRW policy is used. With a local scheduler-based approach each local scheduler determines job priorities independent of other schedulers at other nodes using only information available at its corresponding I/O node. At any instant, the highest priority job at one I/O node may be different from that at another node. A brief description of the scheduling policies is presented next.

The JBNOT ("Not Job Based") policy does not assign priorities to jobs. Instead it services I/O requests on a FCFS basis. This policy is used as a performance yardstick against which the other policies are

compared to see if job based I/O scheduling produces any performance benefits.

The JBOJF ("Job Based-Oldest Job First") policy gives the highest priority to the job which has been running in the system for the longest period of time.

The JBLOIO ("Job Based-Largest Outstanding I/O") policy gives highest priority to the job with the largest total outstanding I/O demand. With a local scheduling approach the total outstanding I/O demand for a job at a given I/O node is the sum of the demand of each I/O request issued by the job for that node that has not yet received service. Because the total outstanding I/O demand varies between I/O nodes and over time, job priorities also vary between I/O nodes and over time. Consequently, the relative priorities of each job must be re-evaluated by an I/O node whenever it becomes idle. For a centralized scheduling approach the total outstanding I/O demand of a job is the sum of the demands for all the requests still enqueued at the global router.

The JBSOIO ("Job Based-Smallest Outstanding I/O") policy is identical to JBLOIO except that highest priority is given to the job with the smallest total outstanding I/O demand. As with JBLOIO, the priority of each job must be re-evaluated whenever an I/O node becomes idle.

The JBRR ("Job Based-Round Robin") policy considers each job in a round robin fashion. If a job has an outstanding request, then the I/O request is serviced by the idle I/O node, otherwise the scheduler looks at the next job. When all the jobs with outstanding I/O demands have been served once the first job will receive consideration once again.

5 The Performance of the I/O Scheduling Policies

The performance of the I/O scheduling policies are presented in this section. A factor-at-a-time approach is used to characterize the performance of these policies: one parameter in the simulation model is varied while the others are held at fixed values. A large number of simulation experiments are run, but to conserve space, only a subset of the experimental results is presented and described. More data is available in [13]. Experiments are run for both the NOP and O1 job models. The fixed valued parameters are included in the legend of each graph that captures the result of an experiment. The rationale for the choice of some of the fixed factors is provided. The number of processors is fixed at 20 because it corresponds to many medium scale shared memory systems that are currently operational. Investigation of an appropriate degree of multiprogramming is beyond the scope of this paper. We have fixed M = 4 which provides an intermediate value that is reasonable for P = 20. On a cluster of workstations for example, this corresponds to four workstations each consisting of a 5-processor shared memory multiprocessor. A number of such commercial-off-the-shelf small scale SMP workstations are available at a reasonable cost from vendors such as Hewlett Packard and Sun Microsystems. The number of I/O nodes is fixed at 4. Experiments with single applications with workload characteristics used in the simulation experiments showed that little incremental improvement in response time accrues from using more than 4 I/O devices. We have run experiments with other values of I and different workload parameters (see [13]).

The results obtained under a centralized data replication strategy (Replicated-C) are first presented, followed by the results from using the distributed data replication strategy (Replicated-D) and from using the FixedRW strategy. Results for the O1 model with Replicated-C presented in Figure 2, show that a small performance improvement is obtained by using I/O scheduling policies based on job characteristics over JBNOT that does not use any knowledge of job characteristics, even as the system approaches saturation (high λ). The results for the NOP workload show that the relative performance of the I/O scheduling policies differ only at high λ, with JBSOIO performing the best followed by JBNOT.

These results indicate that with a centralized data replication policy such as Replicated-C and particularly with the O1 I/O model, the centralized routing of I/O requests to idle I/O nodes has a bigger impact on system performance than using knowledge of job attributes in I/O scheduling. For the NOP workload some performance improvement may be achieved by using a policy based on job characteristics such as JBSOIO but the difference in performance among the different policies diminish with an increase in the variability in I/O demand [13].

Scheduling with the Replicated-D Data Distribution Strategy

Figure 3 shows quite different results when the data replication policy is changed from Replicated-C to Replicated-D. For both O1 and NOP, the JBLOIO policy proves to be the worst while JBSOIO policy proves to be the best policy. Giving a high priority to jobs with high I/O demand can introduce large queueing delays for smaller jobs. The success of JBSOIO suggests that running jobs with smaller I/O demand first can significantly improve the mean job response time. JBOJF policy seems to perform comparably with JBSOIO for an O1 workload, but the second position is taken by JBNOT when the workload is changed to NOP. The JBRR policy performs better than JBLOIO but its performance is inferior to JBSOIO for both the O1 and NOP workloads.

An intuitive justification for the data in Figure 3 is

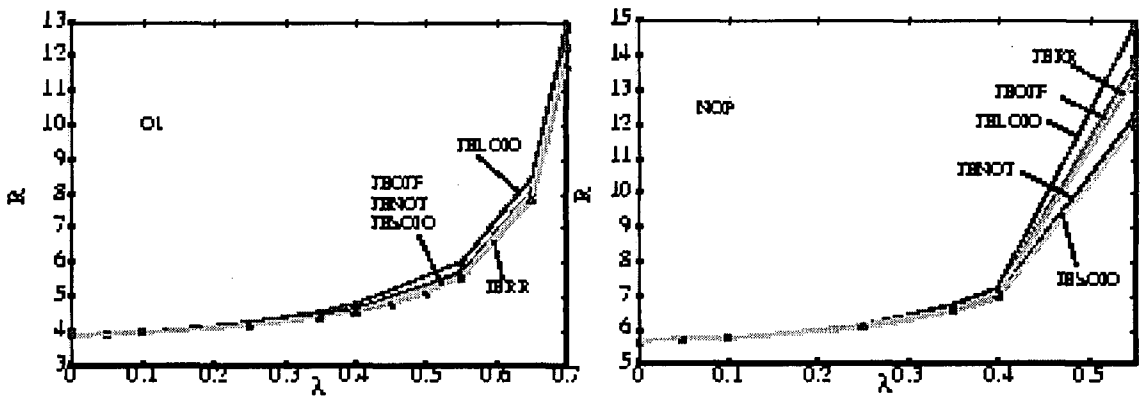


Figure 2: The Performance of Scheduling Policies under Replicated-C. $P = 20$, $I = 4$, $d_s = 0.01$, $CV_s = 1.0$, $F_i = 0.5$, $M = 4$, $n_d = 10$, $d = 1$, $CV_d = 1.0$, $CV_i = 1.0$, $n_i = 5$

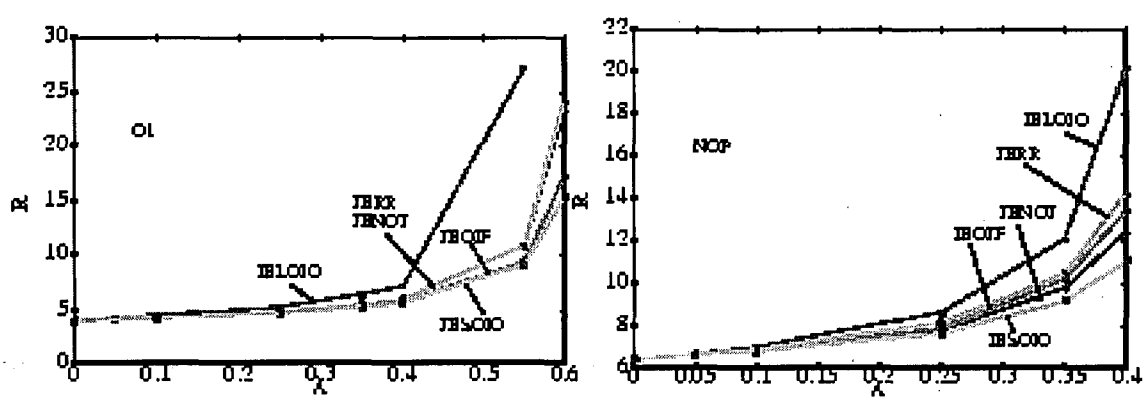


Figure 3: The Performance of Scheduling Policies under Replicated-D. $P = 20$, $I = 4$, $d_s = 0.01$, $CV_s = 1.0$, $F_i = 0.5$, $M = 4$, $n_d = 10$, $d = 1$, $CV_d = 1.0$, $CV_i = 1.0$, $n_i = 5$

presented.

It follows from Little's Law that the smaller the number of unfinished jobs in an open system, the better is the performance of the scheduling policy. One of the reasons behind the success of JBSOIO is its ability to complete jobs quickly by associating a higher priority with jobs that exhibit small I/O demand. The relative performance of JBOJF and JBNOT depends on the I/O model used. For the O1 model all the threads in an application do their initial read operations at the beginning. An I/O request from an older O1 job is likely coming from a thread that has completed its computation and is waiting to write its output. Giving priority to such a job tends to speedup the completion of these nearly completed jobs, which minimizes the number of resident jobs and response time. The situation is different for the NOP model. Irrespective of reads or writes all the I/O requests generated by the jobs in a given phase are generated at the same time. However when a NOP job performs I/O the processors allocated to the job are idle and more than one I/O requests from the same job may be enqueued at a given I/O node. Switching an I/O node to a different job on the basis of its age is detrimental because it keeps the previous job incomplete and all the processors used by the job unused. JBNOT seems to be giving a better performance in this situation. JBRR prevents monopolization of I/O nodes by large jobs and performs better than JBLOIO. But because of the large number of job switching that tends to increase the average number of incomplete jobs on the system its performance is inferior to JBSOIO.

Scheduling with the FixedRW Data Distribution Policy

The performances of the scheduling policies for the FixedRW strategy are presented in Figure 4. The behavior of the policies are similar to the case in which Replicated-D was used as the data distribution strategy. One of the reasons for this similarity is that unlike Replicated-C none of these strategies uses a centralized database for routing the I/O request and multiple requests from the same job may be waiting at the same node while other nodes may be idle. The results for the FixedRW data distribution strategy confirm the utility of using job characteristics in scheduling. It is interesting to note that the performance of JBOJF is comparable to that of JBSOIO for the O1 workload.

6 Conclusions

Based on a simulation model this paper is concerned primarily with effective I/O scheduling strategies for multiprogrammed parallel systems that employ parallel I/O. A number of experiments were run under various types of workload and data distribution strategies. Knowledge gained in terms of insights into system performance and scheduling are summarized.

The Impact of Data Replication: Replication of read data and the flexibility of writing to any free disk seems to improve system performance in a major way when a centralized control is available. The effect of scheduling is only secondary especially for the O1 model.

The Impact of Scheduling: When a distributed routing is used with data replication or when data is not replicated, choosing an appropriate I/O scheduling policy is crucial for achieving high system performance. Using characteristics of jobs such as the total outstanding I/O demand and age seems to be quite effective in I/O scheduling on multiprogrammed parallel systems. A large benefit in performance can be obtained for example by using a policy such as JBSOIO under a variety of workload conditions. JBOJF and JBNOT are also observed to perform well for O1 and NOP respectively. *The Effect of Equal Sharing*: Equal sharing of I/O nodes among applications in the current multiprogramming mix as captured by the Round Robin scheduling policy did not perform well in most situations. Although it prevents I/O starvation experienced by jobs, switching the I/O nodes among a number of jobs seems to increase the number of incomplete jobs and deteriorate mean response time.

Implementation Issues: Although CPU scheduling policies such as Shortest Job First that are based on explicit knowledge of job characteristics demonstrate excellent performance, their exact implementation has been difficult on general purpose systems in which it is hard to acquire such a priori knowledge. In comparison to CPU scheduling, policies based on job characteristics are easier to implement in the context of I/O scheduling. It is possible for the operating system to keep track of the age of a job as well as to estimate the I/O demand associated with a request. Consequently it is possible to implement policies such as JBOJF or JBSOIO on a real system.

Due to limitation of space we could not discuss the experiments conducted for the investigation of the impact of job characteristics on performance. Results of the experiments described in [13] indicate that both the variability in I/O demand and I/O factor affect the relative performance of the scheduling policies in a similar way. For example, an increase in CV_i is observed to produce a larger difference in the performance of the scheduling policies for any given I/O model for both Replicated-D and FixedRW data distribution strategies. A larger benefit accrues from using policies based on job characteristics such as JBSOIO when the I/O factor is increased.

The overhead associated with the scheduling policies as well as with the dispatching of I/O requests have been assumed to be negligibly small in this paper. Measuring these overheads on a real system

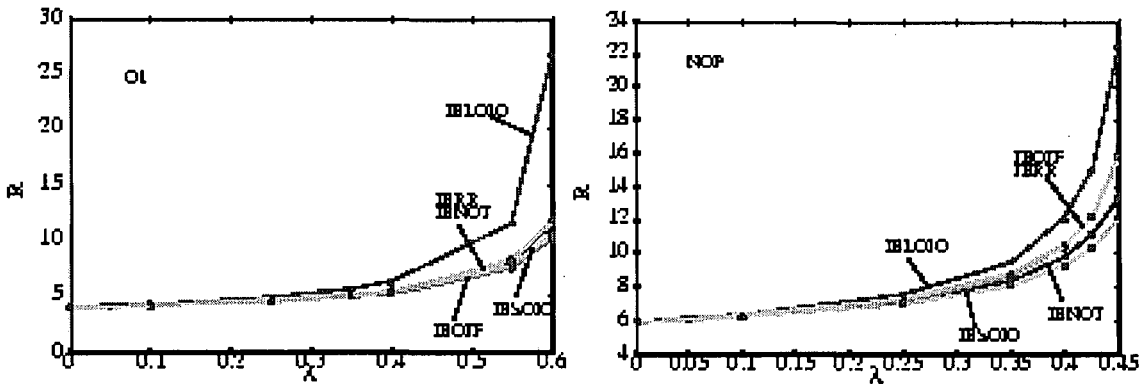


Figure 4: The Performance of Scheduling Policies under FixedRW. $P = 20$, $I = 4$, $d_s = 0.01$, $CV_s = 1.0$, $F_i = 0.5$, $M = 4$, $n_d = 10$, $d = 1$, $CV_d = 1.0$, $CV_i = 1.0$, $n_i = 5$

and understanding their performance impacts is currently being undertaken by the authors. In this paper we have considered systems in which data transfer times dominate the overall disk service time. Systems on which disk seek times can form a significant component of the overall I/O service time warrant investigation. Using the results presented in this paper for implementing scheduling policies on real systems forms an important direction for future research. Work is underway in implementations of these strategies and measuring their performance on a network of workstations.

Acknowledgments
This research was supported by the Natural Sciences and Engineering Research Council of Canada. Thanks are due to Geoff Waddington for proof reading and to the anonymous referees for their comments.

References

[1] A.C. Arpaci-Dussau, D.E. Culler, A.M. Mainwaring, "Scheduling with Implicit Information in Distributed Systems", Proc. ACM SIGMETRICS'98/Performance'98 Joint Conf. on Measurement and Modeling of Computer Systems, Madison, June 1998, pp. 233-243.

[2] S. Chen, D.Townsley, "A Performance Evaluation of RAID Architectures", IEEE Trans. on Computers, Vol. 45, No. 9, October 1996, pp. 116-1130.

[3] Y. Cho, M. Winslett, M. Subramaniam, Y. Chen, S. W. Kuo, K.E. Seamons, "Exploiting Local Data in parallel array I/O on a Network of Workstations", Proc. Fifth Workshop on Input/Output in Parallel & Distributed Computing, 1995.

[4] P.F. Corbett, D.G. Feitelson, J.-P. Prost, S.J. Baylor, "Parallel Access to Files in the Vesta File System", Proc. Supercomputing '93 Conf., 1993, pp. 472-481.

[5] R. Cypher, P. Messina, "Architectural Requirements of Parallel Scientific Applications with Explicit Communication", Proc. International Symp. on Computer Architecture, 1993.

[6] L. Diaconescu, S. Majumdar, "The Performance of Parallel I/O on a Multiprogrammed Network of Workstations", Proc. Parallel and Distributed Computing and Networks, Brisbane (Australia), December 1998, pp. 439-448.

[7] J.M. Del Rosario, A. Choudhary, "High Performance I/O for Massively Parallel Computers", IEEE Computer, March 1994, pp. 59-68.

[8] R. Jain, K. Somalwar, J. Werth, J.C. Browne, "Heuristics for Scheduling I/O Operations", IEEE Trans. on Parallel & Dist. Systems Vol. 8, No. 3, March 1997, pp. 310-320.

[9] Kobayashi, H., "Modeling and Analysis: An Introduction to System Performance Evaluation Methodology", Addison-Wesley, 1981.

[10] D. Kotz, N. Nieuwejaar, et al., "File-Access Characteristics of Parallel Scientific Workloads", Tech Report PCS-TR95-263, Dept. of Math. and Comp. Science, Dartmouth College, Hanover, U.S.A, 1995.

[11] D. Kotz, N. Nieuwejaar, "Flexibility and Performance of Parallel File Systems", Third International Conf. of the Austrian Committee on Parallel Computing (ACPC'96), September 1996, pp. 1-11.

[12] P. Kwong, S. Majumdar, "Study of Data Distribution Strategies for Parallel I/O Management", Third International Conf. of the Austrian Committee on Parallel Computing (ACPC'96), September 1996, pp. 12-23.

[13] P. Kwong, Management of Parallel I/O in Multiprogrammed Parallel Systems, M.Eng. Thesis, Department of Systems and Computer Engineering, Carleton University, 1996.

[14] O. Krieger and M. Stumm, "HFS: a Performance Oriented Flexible File System Based on Building-Block Compositions", Proc. 4th Workshop I/O in Par. and Dist. Systems, Philadelphia (PA) , May 1996, pp. 95-108.

[15] S. Leuttenegger and M. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", Proc. 1990 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, Boulder (CO), May 1990, pp. 226-

[16] S. Majumdar, S., D.L. Eager. and R.B. Bunt, "Characterisation of Programs for Scheduling in Multiprogrammed Parallel Systems", Performance Evaluation, Vol. 13, 1991, pp. 109 -130.

[17] S. Majumdar, Y.M. Leung, "Characterizing Applications with I/O for Processor Scheduling in Multiprogrammed Parallel Systems", Proc. Sixth IEEE Symposium on Parallel and Distributed Processing, Dallas (TX), October 1994, pp. 298-307.

[18] S. Majumdar, F. Shad, "Characterization and Management of I/O in Multiprogrammed Parallel Systems", Proc. Seventh IEEE Symposium on Parallel and Distributed Processing, San Antonio (TX), October 1995,pp. 502-510."

[19] D.A. Patterson, G. Gibson, R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", Proc. ACM SIGMOD Conference, June 1988, pp. 109-116.

[20] Y.N. Patt, "The I/O Subsystem: A Candidate for Improvement", IEEE Computer, March 1994, pp. 15-16.

[21] A.L.N. Reddy, P. Banerjee, "An Evaluation of Multiple-Disk I/O Systems", IEEE Trans. on Computers, Vol. 38, No. 12, December 1989, pp. 1680-1690.

[22] C. Ruemmler, J. Wilkes, "An Introduction to Disk Drive Modeling", IEEE Computer, March 1994, pp. 17-29.

[23] E. Rosti, G. Serazzi, E. Smirni, M.S. Squillante, "The Impact of I/O on Program Behavior and Parallel Scheduling", Proc. ACM SIGMETRICS'98/Performance'98 Joint Conf. on Measurement and Modeling of Computer Systems, Madison , June 1998, pp. 56-65.

[24] W. Stallings, Operating Systems, Internals and Design Principles (Third Edition), Prentice Hall 1998.

Fault Tolerance of Parallel Adaptive Applications in Heterogeneous Systems

D.Kebbal, E.G.Talbi and J.M.Geib
L.I.F.L., Université des Sciences et Technologies de Lille, 59655 Villeneuve d'Ascq Cedex, France
Phone: +333 20 43 45 39, Fax: +333 20 43 65 66
E-mail: {kebbal, talbi, geib}@lifl.fr

Keywords: parallel adaptive programming, heterogeneous systems, checkpointing, fault tolerance

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 13, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

In this paper, we present a fault tolerance approach for managing application faults in parallel adaptive environments. Parallel adaptive systems allow the application to grow as the resources become available and to shrink when these resources are reclaimed or overloaded. Our fault tolerance policy uses an optimized coordinated checkpointing algorithm which allows rolling back the checkpointed applications on heterogeneous architectures and redistributing the load at recovery time. Furthermore, the approach permits to recover from failures by involving a minimum part of the application in the recovery operation after a failure.

1 Introduction

The increasing performance/cost ratio of workstations and fast communication networks have pushed networks of workstations (NOWs) to become popular platforms for parallel and distributed computing. The fault risks increase with the size of the distributed system degrading its performance. Therefore, fault tolerance aspect must be taken into account.

Fault tolerance property of a system is its ability to ensure the continuity of service despite hardware and software faults. This can vary from guarantying a complete service to shutting down properly the system through operating in degrading mode.

Fault tolerance techniques have been largely studied in the literature [1, 2]. They are based on hardware or software solutions and they are implemented differently in the case of uniprocessor, massively parallel or distributed systems. Several software models used to construct reliable distributed computing systems have been proposed [3], among them group-oriented models, transaction models and models based on communicating processes with point-to-point communication. These paradigms include object/action model, primary/backup model, state machine approach and conversations [4]. These techniques are generally based on low-level services that provide functionality similar to standard hardware or operating system services with improved semantics. These services include stable storage, atomic actions, resilient processes and some kinds of RPC [4]. Furthermore, other type of low-level services provide consistent information to all processors in a distributed system. They include common global time, group-oriented multicast and membership services.

The well-known software solutions for the point-to-point communication models are those based on checkpointing/roll-back techniques which consist on periodically defining a consistent global state of the application and saving it on a stable storage. When a failure occurs on a node, the application is rolled back from its last checkpoint [5]. The low-level services used by these mechanisms include particularly stable storage and resilient processes services.

Most of the parallel and distributed programming environments (PVM, MPI, ...) don't handle the fault-tolerance aspect. Moreover, most of the classical checkpointing algorithms developed don't take into account the specific properties of some applications which can be the basis of important optimizations. Furtheremore, They are constrained by the heterogeneity problem: a process checkpointed on a specific architecture must be restarted on the same architecture.

Our purpose is to address fault tolerance in parallel adaptive resource management systems through a system called MARS. Parallel adaptive systems have the particularity of adapting the application load to the resource availability provided by the underlying environments [6, 7, 8].

The remainder of the paper is organized as follows: the following section presents some similar systems. Section 3 introduces the MARS system and parallel adaptive programming. In section 4, we describe

briefly an algorithm for checkpointing parallel adaptive applications. Section 5 presents a complete fault tolerance approach used to manage failures in the system. Thus, a new programming interface is developed in order to handle work automatically and to give a framework for recovering only the failed application components.

2 Related work

In the group-oriented models, the consistency is guaranteed by some mechanisms like identical process group views and delivery order of messages. Thus we can distinguish global order in which a lockstep mode is used and a *virtually synchronous model* introduced by Isis [3] in which only a causal order required to keep the consistency of operations is used.

Horus [9] and Relacs [10] uses the same model but deals with multiple partitions of the same group created after network partition. Multiple views of the same group may therefore exist concurrently. Thus, new operations are added including partition merge and state transfer. Moreover, [10] uses the notion of unreachability to keep track of possibly created partitions.

In Horus, a flush protocol is used when a failure is detected or a merge operation is initiated. The protocol consists of re-sending unstable messages [1] to all members in the group before installing a new view of the group. The installed view will include all partitions after a partition merge or exclude a crashed process in the case of failure. This protocol guarantees that all members in the new view deliver the same set of messages (virtualy synchronous model). The stability property is known by keeping at each member a matrix where each entry (i, j) contains the number of messages sent by process i and viewed by j. Based on this matrix, a report including the sequence numbers of the last delivered messages is periodically broadcast to all group members which may incure considerable overhead.

We consider an application model consisting of a set of distributed processes communicating through a local area network, by sending point-to-point messages. The fault tolerance approach uses a checkpointing mechanism and a backward error recovery scheme to recover from failures. The checkpointing/roll-back techniques are well used in commercial applications in which the availability and the performance properties are the well-required properties. In contrast, group and multicast-based models use generally costly mechanisms like replication, multicasting and consensus protocols to ensure consistency of group membership and message delivery.

[1] A message is called stable when it is seen by all members

Checkpointing algorithms have been largely studied (see [1, 5]) and can be classified in three categories [11]:

- Explicit techniques in which the programmer is responsible for the implementation and management of the checkpointing/roll-back mechanism [12]. Some application properties are used to achieve efficient implementation with increased programming complexity.
- Semi-automatic techniques which reduce this complexity by providing checkpointing and roll-back libraries, but they leave the management of checkpointing to the programmer [13].
- In the user-transparent techniques, checkpointing implementation and management are ensured by the library. We can distinguish three sub-classes: independent techniques in which checkpoints are taken independently by each application process [14]. Consistent checkpointing approches in which the whole application is frozen at checkpoint time and all processes cooperate to define a consistent recovery line [15, 5]. The third sub-class called hybrid techniques tries to combine the advantages of the two previous categories in order to achieve better performance [16].

The consistent checkpointing algorithms result generally in efficient checkpointing implementation and management. They avoid orphan and duplicated messages and don't require determinism. In contrast, the synchronization phase involved increases the checkpointing cost and latency.

Some systems based on consistent checkpointing have been developed around the PVM environment [17]. A checkpointing algorithm for the MIST system is presented in [18]. Its complexity in terms of control messages exchanged at checkpoint time is $O(n^2)$. It handles considerable amounts of data since it considers the address space of all application processes. Fail-safe PVM [19] and CoCheck [20] use an algorithm similar in complexity. Fail-safe includes the PVM daemons in checkpointing. Moreover, they are concerned by the heterogeneity problem since they include the address space of all application processes in the checkpoint file.

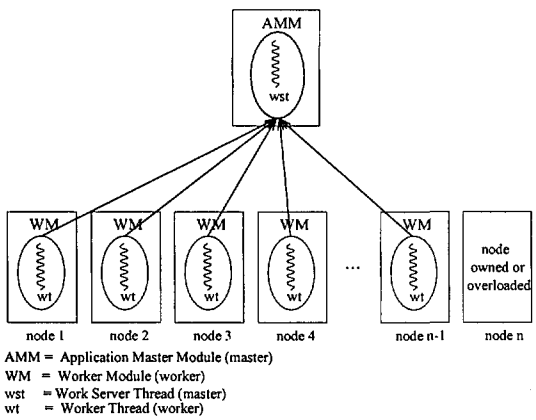
3 Adaptive programming

Adaptive applications are those applications in which the parallelism degree can be managed dynamically during execution time. The application can adapt its size to the computing load provided by the underlying environment. The ownership aspect of workstations is respected without overloading other nodes in the system and the idle cycles are efficiently used. Piranha

[6], MARS [7] and CARMI [8] are examples of such systems.

In MARS, the computation is divided into some work units managed by a master process and allocated dynamically to some workers. At the master level, the programmer specifies a work server module (thread) responsible for allocating work and receiving results from workers. The worker body is generally a loop consisting of three sequential steps: getting a work unit from the master, processing it and putting back results (figure 1). If a workstation is reclaimed by its owner or becomes overloaded, the system *folds up* the application by withdrawing the worker on this workstation. The evacuated worker puts back its pending work to the master before dying. In contrast, if a workstation becomes idle, the system *unfolds* the application by creating a new worker on the idle node.

Figure 1: Structure of a MARS application



MARS is built on a multithreaded parallel environment called PM2 [21] which uses PVM [17] as communication support.

4 Checkpointing parallel adaptive applications

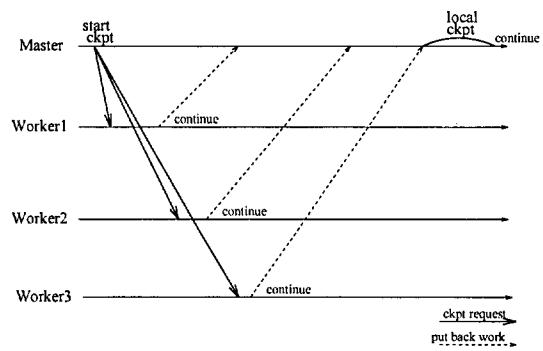
The checkpointing algorithm is based on "consistent checkpointing" in the sens that it involves all adaptive application components in the checkpointing operation. However, it differs from the algorithms of this category in three points: (i) It doesn't include the whole workers address spaces in the checkpoint file but only the work units they are processing (partial work). (ii) Since the workers don't communicate directly but only with the master, the algorithm uses only $O(n)$ messages to synchronize the workers instead of $O(n^2)$ required generally by consistent checkpointing algorithms. (iii) The workers resume their execution immediately after putting back the pending work.

4.1 Checkpointing

The checkpointing algorithm is described below (figure 2):

1. When a checkpointing operation is started, the master broadcasts a request to all its current workers in order to put back their partial results.
2. On receiving this request, each worker suspends its execution and starts a specific control thread which puts back the partial results to the master. The control thread is synchronized with the computation threads in order to define a consistent local state of the worker.
3. After receiving partial results from all its workers, the master takes its local checkpoint. The single Unix process checkpointing tool of Condor is used [22]. Before it takes its local checkpoint, the master forks a new process which takes the local checkpoint in parallel with the execution of the application and dies. Indeed, this is similar to the main-memory optimization except the overhead induced by the process creation which is higher than the thread creation in main-memory techniques.

Figure 2: Checkpointing algorithm at work



A detailed presentation and performance evaluation of the checkpointing/roll-back algorithm can be found in [11].

4.2 Roll-back

Rolling back the application consists in the case of the master failure of getting back the application state from the checkpoint file. Note that the determinism is not required since the checkpoint defines a consistent

recovery line and there are no logged messages. The rolled back master reenrolls in the MARS run-time system as a new application without workers. The work units partially processed is redistributed dynamically to the new workers created after requesting the system for available resources at reenrollment time.

The number of new workers will be function of the current load which can be different from their number at failure time.

4.3 Experimental results

In order to give an idea on the performance of the checkpointing algorithm, we have experienced the mechanism using two applications: *tabu*, a parallel version of tabu search program used to solve a QAP[2] problem and *prime* a parallel program which finds prime numbers belonging to a given range. Table 1 recapitulates the running time and the checkpointing overhead induced by the mechanism with 5 *mn* checkpointing period using a network of 13 workstations. The results show that the checkpointing overhead of the master in negligible (under 1%). The second part of the table 1 presents average values of workers measurements which do the effective work. The results show that the checkpointing overhead is under 0.5% for both applications.

In contrast, the size of the single checkpoint file increases with the number of workers (partial work). Thus, the mean checkpoint file size reaches 1.7 *Mbytes* with a standard deviation of 76 *Kbytes* for *tabu search* and 3.4 *Mbytes* with a standard deviation of 1300 *Kbytes* for the *prime* program. Fortunately, the main-memory technique used minimizes the impact of the file size on the checkpointing performance by allowing the application to resume its execution while the file is being written on the stable storage.

4.4 Qualitative analysis

Our approach presents a number of advantages:

- Load balancing: At recovery time, the application is rolled-back by restarting the new master from the checkpoint file without workers. The load can be balanced by selecting the under-loaded nodes to start the new workers.
- Heterogeneity: In classical algorithms, rolling back a process checkpointed on a specific architecture must be done on the same architecture. A restored application can't benefit at the highest degree from the availability of the system. Since we don't deal with workers address space, the application except the master can be restarted on any available nodes whatever the architecture.
- Independence of the system load at recovery time: Once an application is restarted, it doesn't require that the resources number be the same as at checkpoint time. Indeed, it can restart with less or more workers.
- Transparency: The checkpoint algorithm doesn't require additional user interventions since it uses the *fold* service used by the MARS run-time system itself. An API is provided to the user for explicitly managing the checkpoint mechanism.
- Portability: The checkpointing algorithm is implemented at user level, using facilities available through standard Unix system calls and libraries. The advantage of such an implementation is that there is no need for kernel modification, making it portable to various Unix flavors (SunOS, Solaris, OSF and Linux).

Unfortunately, the approach suffers from some drawbacks especially the overloading of the master at checkpoint time, since it receives pending work units from all workers at the same time before taking its local checkpoint.

Our approach has been used in coarse grained applications in scientific computing (Gauss elimination, matrix multiplication, etc) and combinatorial optimization (branch and bound, heuristics, etc) fields.

5 Failure detection and recovery

We assume a distributed asynchronous system in which the relative processors speed and the communication delay are not known a priori. The underlying communication system ensures the sequencing and the delivery of messages resulting in FIFO channels.

Processes fail by crashing (by prematurely halting) allowing then other components to detect the failure. Failure detection is achieved by superimposing to each process a failure detector [23] in which a "timely" failure detection approach is used.

In Isis [3], the communication transport layer is integrated with the failure detection layer to make processes appear to fail by halting (fail-stop model). The system uses an agreement protocol to maintain a list of non faulty processes. However, a process suspected to be failed is forced to leave the system.

As we will see it in the failure recovery section, the processes can belong to one of two classes: system and application processes. The system processes consist of a set of communication and resource management daemons. The application processes consist of user application processes. When a communication or a resource management process crashes or is suspected

[2]Quadratic Assignment Problem

Table 1: Running time and checkpointing overhead

Application	Without Ckpt (s)	Checkp Overh (s)	%	Checkp number	Mean Number Workers/Ckpt	Mean Ckeckp Overhead (s)	%
tabou	18821	127	0.67	63	10.96	1.05	0.32
prime	12434	77	0.62	42	11.76	1.26	0.40

to be crashed, the node that it controls is considered entirely failed even this is caused by the interruption of the communication channel. In contrast, the failure of an application process may affect only that process [3]. Following this classification, the failure detection approach used in our model is achieved in two ways: first, by requesting the communication layer [4] to inform some system component when a specified process stops its execution. This is especially useful for application processes when the crash is not due to a real failure of a node. Second, by using a timeout-based mechanism in which the processes are periodically "pinged" so that a failure of a process is detected quickly. Later if a process which was running on the failed node interacts with the system, it is forced to leave the system. This choice is motivated by the fact that these failures can take excessive time, preventing thus the application components depending on the computation being done by some unreachable processes from progress. Our adaptive system provides tools for adding new available nodes to the system. However, the computation being done on the unreachable node can be ignored and restarted from a previously saved state on another idle node. Therefore, progress is usually guaranteed. One might think that this approach raises a problem with transient failures caused by temporary unavailability of the process–overloading of the node for example–. In practice, the timeout period is set sufficiently high to avoid such transient problems. The first detection approach is guaranteed by a communication daemon running on the specified node which knows immediately for the failure of the local processes. If such a failure is registered, the requesting process is informed. However, the failure detection in this case is fast and more efficient. The approach can be combined with a probabilistic failure interval ditribution to adapt the checking period so that the communication cost can be minimized.

Subsequently, we distinguish three types of failures: the *system failure*, in this case all applications must be terminated, the system restarted and the applications rolled back manually. The *master failure* which involves rolling back an entire application. The third type of failures is the failure of a specific worker which is recovered by rolling back the failed worker only. To allow partial recovery, the system must keep track of all tasks assigned to each worker. However, a transparent management of work is required.

[3]The failure can be caused by a conceptual or runtime error like memory access violation or external intrusions like explicit interruption of the process by the user

[4]PVM communication library

5.1 Managing work transparently: A framework for partial recovery

In order to allow a transparent management of the computation, a programming interface based on dependency graphs model is developed. The user can therefore build easily his parallel application. The application construction consists of specifying the application tasks and describing the precedence constraints between them.

A task is a data fragment on which some sequential actions must be applied. At the master level, the user specifies his tasks, packs them into data structures and gives them to the library using a specific function. Then, the library builds a work space (bag of tasks) and a dependency graph which ensures the execution order of tasks when allocating them.

The allocation of tasks to workers is achieved automatically by a library module (the work server module) without user intervention except that the user must specify how his data is packed and unpacked with PVM functions. The user can interpose also when receiving pending work and results for a possible optimization of storage space or execution time or for generating new tasks on-line.

The body of the worker consists generally in a loop which begins with a library call interpreted by a transparent message sent to the master. The work server searches in the master work space for a ready task. When it find a task, it returns it to the worker including some other information: the name of the thread designed to process the task and a flag specifying if the task has been partially processed (pending work) or it is simply a new task (new work).

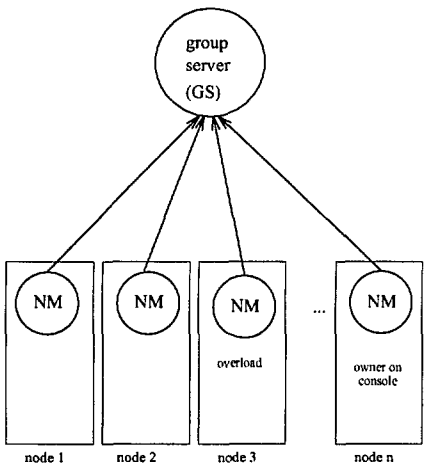
Getting the task and the thread name, the specified worker thread can be started. When it finishes, it calls a specific function which resturns the results to the master.

When the system decides to fold up the worker or a checkpointing operation is started, a specific user function is called which puts back the pending work and the partial results to the master.

5.2 How recovery is achieved

The MARS system consists of a *"group server"* (GS) which manages a pool of heterogeneous workstations. On each node, resides a *"node manager"* (NM) which informs systematically the GS of the node state transitions (figure 3).

Figure 3: Architecture of MARS



When the application enrolls into the system, it starts a specific process called the mediator on the node where the GS is running. Its roll is managing all application "group server" communication and rolling-back the application on failure [5].

The fault tolerance algorithms must ensure efficient fault detection and management as well as the consistency of the fault tolerance policy by avoiding the recovery of some application failures caused by conceptual or human errors.

5.2.1 Recovery at system level

The failure detection at this level is achieved following the first model described above. Thus, the GS keeps a *failure detector* which looks for *node managers* failures by "pinging" them periodically. When a *node manager* failure is detected, the node is declared failed and all processes that it hosts are considered crashed. Therefore, the GS takes the following actions:

Node failure: When a node is declared failed, the GS is notified. Thus, it tries to recover all the applications where the master was running on the failed node as follows:

- It tries to roll back the application entirely. This is done by calling the mediator which, giving it a node identifier of the same architecture as the failed one [6] within the available nodes, it rolls back the application that it controls.
- If the roll-back is not achieved, some other restoration attempts [7] are made. If after all attempts, the application is not rolled back, an error is suspected, the recovery is aborted and the mediator is terminated.
- If the application is rolled back after one or some attempts, some information about the failed application (its identity and the incarnation number of the master) are kept in the table entry of the failed node. This information is used by the GS to distinguish an application termination due to a node failure from an abnormal behaviour of the application (see the section on the master failure at the application level below).

Concurrently with the applications roll-back, the GS tries to recover the failed node within a finite time. Otherwise, it reconfigures the system by deleting the failed node from the configuration.

The application recovery is made immediately without waiting the failed node to be available. We proceed so to avoid the eventual waiting time taken by repairing the failed node.

Master termination The master failure can be seen by the mediator via the communication layer as described above or after the expiration of a timeout period when communicating with the master. Therefore, a *failure message* is sent to the GS which executes the recovery protocol:

- If the termination is not due to the failure of a recovery attempt, the node hosting the master is checked. If it is really failed, the master is searched and freed from the failed masters list of the down node using the information kept previously.
- In contrast, if the node is alive or the master is not found in the down node list, the system suspects a conceptual or human error and the application is simply terminated. This procedure is executed to ensure a consistent state of the system. The application recovery is invoked when the node failure is revealed as indicated above.

Note that, the normal termination of the application is distinguished from the master failure by a *termination message* sent by the master to the mediator

[5] Our goal is to keep a process (the mediator) with the same rights as the application and which can survive when the whole application crashes. This is motivated by the fact that the system hasn't root privileges and the application must not execute in root mode even the system is root

[6] This is constrained by the master roll-back which must be done on the same architecture as the checkpointing one

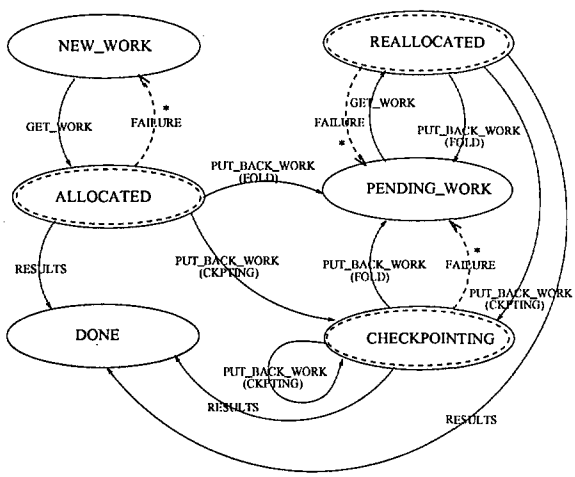
[7] three attempts in the current implementation

and forwarded to the GS. The application entry in the applications list is simply freed and the recovery procedure is not runned.

5.2.2 Recovery at application level

In a MARS application, the work server module allocates tasks to workers and keeps their state. Figure 4 shows the transitions of a task state following the different events.

Figure 4: Transition graph of a task and workers recovery



A task assigned to a worker can be in three states: *ALLOCATED*, when it is allocated for the first time, *REALLOCATED* for a task making the object of more than one allocation. The *CHECKPOINTING* state is the state of a task allocated to a worker being checkpointed. When an application worker is *folded*, it puts back the pending work before dying. Thus, the work treated partially of each task kept by the failed work is stored in the task *pending_work* structure and its state moves to *PENDING_WORK*. However, the work made previously by the evacuated worker can be exploited.

Again the failure detection of workers is achieved at the master level by either the information provided by the communication layer, by timeouts when communicating with workers or explicitely by the GS when it detects a node failure. The workers recovery is achieved by altering the tasks state. Indeed, the failures can be viewed and managed at this level as follows:

Worker failure: When a worker terminates, the master is informed. However, to distingish a normal termination from a failure, the master keeps for each worker a flag which reflects at any time the worker state viewed by the master. If the worker terminates while it is not in a normal termination state, a node failure or a human error is suspected. Then, the worker is recovered.

The recovery of a worker is achieved by browsing the master work space serching for all tasks assigned to the failed worker and altering their state as shown on figure 4. Thus, if the task was in the *CHECKPOINTING* or in the *REALLOCATED* states, its *pending_work* structure is full. Therefore, it moves to *PENDING_WORK* state. In contrast, if it was in *ALLOCATED* state, all the work made previously is lost (the *pending_work* structure is empty) and it becomes in the *NEW_WORK* state.

Master failure: If the master task fails, the following actions are taken:

- All application workers are informed by the communication layer or when they "timeout" when communicating with the master. The informed workers terminate immediately. Thus, we ensure that there are no useless or duplicated processes after the application recovery.
- The mediator failure detector knows about and informs the GS. As it has been presented, the GS determines whether the termination is due to a node failure in which case the recovery procedure is invoked. Otherwise, the application is terminated. If the GS decides to recover, the mediator starts the new master which reenrolls into the system as described in section 4.2.

Mediator failure: The mediator failure is detected by the master and interpreted as the failure of the GS which means that the system is going down. Therefore, the application must terminate. The master proceeds to terminate its workers before ending.

5.3 Advantages of the approach

Although this new programming model is required for managing failures transparently, the approach presents some other advantages: i) a decreased programming complexity by hiding some distributed aspects to the user (communication, synchronisation and work allocation), ii) transparency of failure detection and management and iii) efficiency of the recovery scheme by involving only the failed workers in the roll-back procedure.

6 Conclusion and future work

The grow of parallel applications requirements and the technology evolution have pushed NOWs to become

popular platfoms for parallel and distributed computing. Such environments are shared between many users and the failures frequency is important.

Most of the parallel and distributed programming environments (PVM, MPI, ...) don't handle the fault-tolerance aspect. Moreover, most of the classical checkpointing algorithms developed are constrained by the heterogeneity problem.

Our algorithm exploits the adaptive application characteristics to reduce the complexity of the checkpointing algorithm both in terms of control messages exchanged at checkpoint time which is $O(n)$ instead of $O(n^2)$ involved by classical algorithms and in terms of storage space required by the checkpoint file. Moreover, it solves the heterogeneity problem and allows the load redistribution at recovery time.

The major part of existing checkpointing algorithms are not used in distributed systems. Therefore, the failure detection and management aspects are not discussed. Our fault tolerance approach uses simple, efficient and practical tools for dealing with failures while keeping the consistency. Although the model used seems sometimes simple, it avoids excessive overhead incured by some fault-tolerant mechanisms like multicasting, replication and total order of message delivery. In addition, the approach presents some advantages:

- A maximum of transparency in detecting and managing failures.
- An efficient recovery policy by rolling back only the failed application components.
- A programming interface which provides a simple parallel programming methodology and gives a support for scheduling applications in adaptive environments through keeping track of application tasks and their precedence constraints.

References

[1] G. Deconinck, J. Vounckx, R. Cuyvers, and R. Lauwereins. Survey of Checkpointing and Rollback techniques. Technical Report 03.1.8 and 03.1.12, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, Belgium, Jun 1993.

[2] Observatoire Franąis des Techniques Avancées. *Informatique Tolérante aux Fautes.* Masson, Paris, France, 1994.

[3] K. P. Birman. *Building secure and reliable network applications.* Manning Publications Co, 1996.

[4] S. Mishra and R. D. Schlichting. Abstractions for Constructing Dependable Distributed Systems. Technical Report TR92-19, Department of Computer Science, The university of Arizona, Aug 1992.

[5] J.S. Plank. *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton University, 1993.

[6] D. L. Kaminsky. *Adaptive Parallelism with Piranha.* PhD thesis, Yale University, 1994.

[7] Z. Hafidi, E-G. Talbi, and J-M. Geib. MARS: Adaptive scheduling of parallel applications in a multi-user heterogeneous environment. In *ESPPE'96 proceedings, Alpe d'Huez, France*, pages 119–122, April 1996.

[8] J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[9] R. V. Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communication System. Technical Report TR 94-1442, Department of Computer Science, Cornell University, Aug 1994.

[10] Özalp Babaoğlu, R. Davoli, and A. Montresor. Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications. Technical Report TR UBLCS-95-18, Department of Computer Science, University of Bologna, Italy, Nov 1995.

[11] D. Kebbal, E. G. Talbi, and J. M. Geib. A new approach for checkpointing parallel applications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1643–1651, Las Vegas, Nevada, USA, June 1997.

[12] J. S. Plank, Y. Kim, and J. J. Dongarra. Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations. *Research Report UT-CS-94-268, Department of Computer Science, Universite of Tennessee, Mathematical Science Section, Oak Ridge National Laboratory*, Dec 1994.

[13] J. Maier. Pact- A Fault Tolerant Parallel Programming Environment. In *1st International Workshop 'Software for Multiprocessors and Supercomputers: Theory, Practice, Experience' SMS TPE 93*, St Petersburg, Russia, Feb 1993.

[14] E. N. Elnozahi and W. Zwaenepoel. On the Use and Implementation of Message Logging. In *24th International Symposium on Fault-Tolerant Computing*, pages 298–307, Jun 1994.

[15] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63-75, Feb 1985.

[16] J. Xu and R. H. B. Netzer. Adaptive Independent Checkpointing for Reducing Rollback Propagation. In *15th IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, Dec 1993.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mauchek, and V. Sundernam. PVM: A User's Guide and Tutorial Networked Parallel Computing. *The MIT Press Cambridge, Massachusetts London England*, 1994.

[18] J.Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with Transparent Migration and Checkpointing. *Presented at the 3rd Annual PVM user's Group Meeting, Pittsburgh, PA*, May 7-9 1995.

[19] J. Leon, A. L. Fisher, and P. Steenkiste. Fail-Safe PVM: A Portable Package for Distributed Programming with Transparent Recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University Pittsburgh, February 1993.

[20] G. Stellner. Ressource Management and Checkpointing for PVM. In *2nd European User's Group Meeting*, pages 131–136, Sep 1995.

[21] R. Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irréguliéres.* PhD thesis, L.I.F.L., Université des Sciences et Technologies de Lille, France, 1996.

[22] T. Tannenbaum and M. Litzkow. Checkpointing and Migration of Unix Processes in the Condor Distributed Processing System. *Dr. Dobbs Journal*, pages 40-48, Feb 1995.

[23] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. In *The Tenth ACM Symposium on Principles of Distributed Computing*, pages 325-340. ACM press, Aug 1991.

Fault tolerant execution of Compute-intensive Distributed Applications in LiPS [1]

Thomas Setz
Technische Universität Darmstadt
Alexanderstr.10, D-64283 Darmstadt, Germany
email thsetz@acm.org

Keywords: Hypercomputing, Linda, Software fault tolerance, Workstation Cluster Computing

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 3, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

This paper illustrates how fault-tolerant distributed applications are implemented within LiPS *[SF96, SL97, T.99], a system for distributed computing using idle-cycles in networks of workstation. The* LiPS *system employs the tuple space programming paradigm, as originally used in the* Linda[2] *programming language. Additionally, applications are enabled to terminate successfully in spite of failing nodes. The used mechanisms are transparent to the application programmer and assume deterministic application behavior.*
The implementation is based on periodically writing checkpoints, freezing the state of a computational process, and keeping track of messages exchanged between checkpoints in a message log. The message log is kept within the tuple space machine implementing the tuple space and replayed if an application process recovers. This allows for independent generation of - and recovery from a checkpoint. The approach alleviates the need for application-wide synchronization in order to generate sets of consistent checkpoints and avoids cascading rollback due to the domino-effect. As a result of our approach, applications are able to adapt smoothly to changes in the availability of used workstations.

1 Overview

Workstation computers are becoming increasingly popular due to their high performance/cost ratio. With increasing numbers of workstations and the advent of high-speed networks, supercomputer-like aggregate computational power is available and can be used to perform useful computations.

Application programmers working in this environment must be provided with a programming system facilitating the development of distributed applications. This is accomplished by mechanisms shielding the programmer from the complexity of system-level programming, thus enabling him to concentrate on solving application-level problems. For example, a heterogeneous environment of different operating systems, network protocols or processor architectures should be hidden from the programmer. Implementing a distributed application is also made more difficult by frequent changes in the availability of nodes and networks.

Transparency, meaning hiding the physical implementation of a distributed application, is the utmost goal of every distributed programming system.

The LiPS system enables users to implement distributed applications in heterogeneous networks of workstations, connecting machines with different processor architectures and Unix[3] operating system flavors. The system ensures that only workstations which are considered idle by their users are used within the distributed computations. The system also guarantees successful completion of distributed computations in spite of failing machines or network links. Within the last years, LiPS has been used to distribute computations on about 250 workstations connected to the campus network at the University of Saarbrücken (Germany) and will be enhanced to distribute applications on more than 1000 machines within the next years.

This paper presents the basic decisions taken when designing LiPS version 2.4. This version supports a software-fault-tolerant generative communication paradigm based on the tuple space, as introduced by the coordination language Linda [GCCC85].

The next chapter contains an introduction to generative communication, a programming paradigm suited to implement distributed computations in networks of workstations. Then, an introduction to the terminology used for coping with fault-tolerance is given along with the model of software failure patterns used throughout this paper. Next, the concept of software

[3] Unix is licensed exclusively through X/Open Company Limited

fault-tolerance is introduced. It permits grouping software components and strategies into layers thus supplying general distributed applications with a variety of software fault-tolerance mechanisms. This section also illustrates different methods to restart single crashed processes within the application framework and discusses the benefits of our approach. The general concept of software fault-tolerance is then applied to distributed applications implemented along the generative communication paradigm. Using layer-3 software fault-tolerance enables the user to transparently implement fault-tolerant applications. By relaying all communication activities via the tuple space, a complete log of all messages exchanged between application processes is available in the tuple space machine, even while individual application processes are prone to failure. Having access to a complete history of messages exchanged per process permits recovering application processes in a highly efficient manner, but this requires a well-suited design for the tuple-space-machine. The last section presents the design of our Fault-Tolerant Tuple Space Machine [Set96, Set97] along with its integration into the LiPS system.

2 Related work

There are different approaches to integrate different levels of fault-tolerance into tuple space based applications. Following [BDE94], these approaches can be divided into extensions to the tuple space runtime system [Xu 88, LX89, CKM92, PTHR93] making tuple space fault-tolerant, resilient data and processes [KS90, KS91] making tuple space and processes working on it recoverable, and transaction based or transaction style like language extensions enabling the programmer to define a sequence of tuple space operations as an atomic operation which will be evaluated completely or not at all [BDE94, BS93].

In LiPS version 2.4 we follow the approach to resilient data and processes and integrate the needed mechanisms into the tuple-space-machine. Two runtime-systems, based on the same mechanisms are used to integrate fault-tolerance on the system- as on the application level.

3 Generative Communication

In order to implement distributed applications, a programmer must be supplied with primitives enabling him to create additional processes or tasks, and to exchange messages among them. A conventional programming language, when augmented by inter-process communication and process manipulation primitives, is sufficient for implementing distributed algorithms. Interprocess communication (IPC) may be established accessing the network protocols, using systems like PVM. Another approach, which is used throughout this work, is to use higher level paradigms as the Tuple space based generative communication [Gel85]. These approaches differ with respect to usability, efficiency, and availability on different platforms. While IPC using direct access to network protocols permits highly efficient communication, applications implemented using this approach are rather cumbersome to maintain. The generative communication approach to IPC trades efficiency against ease of use, due to the overhead introduced by Tuple space management. This overhead may be kept down to a reasonable amount by analyzing communication patterns at compile-time.

This section describes the Tuple space based generative communication paradigm. Using this paradigm yields elegant solutions for communication patterns typically found in distributed applications.

3.1 The Tuple Space

The Tuple space is an associative, shared memory accessible to all application processes. It is called associative as it contains data tuples which may be retrieved addressable by their contents rather than by physical addresses, using a pattern-matching mechanism. The implementation of Tuple space memory is hidden from the user and therefore may be realized on a shared-memory machine, a tightly-coupled parallel computer, or on a network of workstations. Data tuples consist of a list of simple data types. We distinguish active tuples generated with the `eval()` operator from passive tuples generated with `out()`. Active tuples are used to create new threads of control within a distributed application while passive tuples are merely used to store data items. A set of operations (`in()`, `rd()`, `inp()`, `rdp()`) is used to retrieve passive tuples. Both blocking and non-blocking versions of tuple retrieval functions are available. Hence, these operations may be used for synchronization *and* communication tasks. The tuple extracting operations `in()` and `inp()` read a data tuple and remove it from the Tuple space . If no tuple is available, the non-blocking operation `inp()` immediately returns an error as opposed to the blocking operation `in()` which suspends the calling thread until such a tuple is found. The tuple reading operations `rd()` and `rdp()` return a data tuple, again in a blocking and non-blocking manner, but do not extract the tuple from the Tuple space . A more elaborate description of the Tuple space can be found in [Set96].

3.2 Benefits of the Generative Communication

As the Tuple space is conceptually separated from an application process, its content is not lost across thread exits. Data tuples remain available until they

are consumed by some other process, which must not necessarily be around at the time the tuple is created. As a result, inter-process communication is decoupled in time. As data tuples are identified solely by their contents, and not by any other means such as senders' or recipients' process-ID, communication is made "anonymous", in that communicating processes do not need knowledge of their peers' identity. IPC using the Tuple space thus decouples communicating processes logically and physically. This eases application development when compared to using a message-passing based paradigm.

As processes in a distributed application have no notion of a peer's location, migrating processes in the case that a machine becomes unavailable due to load increase or crash is made easier. A process may still retrieve messages even when it has to change to a different machine. This mechanism is transparent to the application programmer as no host addresses are involved. The Tuple space communication paradigm is not tied to a particular programming language, hardware or software environment. It may thus be used for distributed applications running on a heterogeneous set of workstations. The paradigm also allows for adapting the number of usable machines, implementing what is called "adaptive parallelism" in [CFGK94, GK92]. Applications may use all available machines, shrink down to the usage of only one, and switch between these bounds of possibilities very easily. Finally, integrating the Tuple space based generative communication approach into a conventional programming language requires only six additional operations.

Therefore, Tuple space based applications turn out to be an adequate choice for implementing distributed applications running on networks of workstations.

4 Failure Models

Workstation computers are prone to failures. As a consequence, this may lead to failures in applications implemented using the LiPS system. Several failure patterns can be distinguished:

- Crash-failures or as called in [SS83] "fail-stop-processors", are observed when a machine halts on an error condition, forcibly terminating all application processes local to the processor affected.
- Soft-fail-stop-failures are observed when a machine stops on an error, terminating all local application processes. But there exists storage, possibly residing on another unaffected machine which remains intact and is accessible.
- Omission-failures are observed when machines sometimes fail to send or receive messages.
- Byzantine failures, where machine start sending wrong and even contradictory information as a result of an error.

The LiPS system is able to cope with soft-fail-stop failures. Data that should remain accessible in spite of machine failures is kept in a storage called repository. The system further deals with omission-failures as messages are exchanged using the UDP protocol of the TCP/IP protocol suite[4]. Handling Byzantine failure is rather expensive, and these failures are rarely observed in practice. Therefore, we will not consider Byzantine failures in this work.

5 Software Fault-Tolerance

Distributed applications are usually based on fault-tolerance mechanisms provided by a node operating system. The term "software fault-tolerance", as introduced in [YC83], is used to subsume methods and software components responsible for detecting and correcting errors causing a distribute application to crash or hang, that are not already handled in the underlying operating system. Software fault-tolerance may be organized in layers – Figure 1 gives an overview. Layers are discriminated along the levels of availability and data consistency.

Normally, distributed applications are based on the services delivered by the node operating system, the so-called level 0 of software fault-tolerance. If a node crashes, manual intervention is required to restart the processes which were residing on that node. Shared data may be lost or left in an inconsistent state.

Layer-1 software fault-tolerance is reached by providing for automatic restart of application processes in the event of a crash. This layer provides for enhanced application availability, as no manual intervention is required for the entire application to complete. Restarted processes still need to re-do their entire computation, resulting in a complete loss of effort spent on the previous run. Abort of a single process may force the entire application to halt if shared global data is left in an inconsistent state, thus wasting the entire time spent computing so far.

Layer-2 software fault-tolerance requires application processes to create checkpoints capturing a process's state. If an application process crashes, it can be restarted from its latest checkpoint, thereby reducing run time spent as effort to reach the state at crash time. Furthermore, messages sent and received in the interval between checkpoint generation are kept in a message log. If an application process restarts from a checkpoint, it will receive the same set of messages it got on its initial run and therefore will compute the

[4]This protocol implements a "best-effort" delivery. Datagram messages may be lost or duplicated by the underlying network layers.

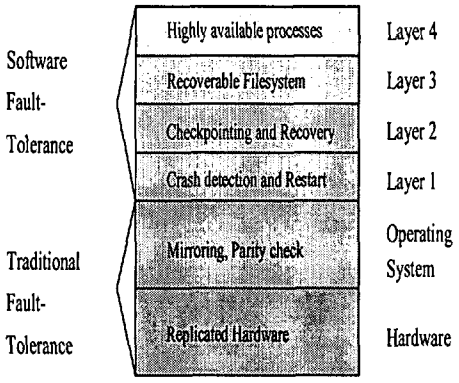


Figure 1: Layering of Software Fault-Tolerance Strategies

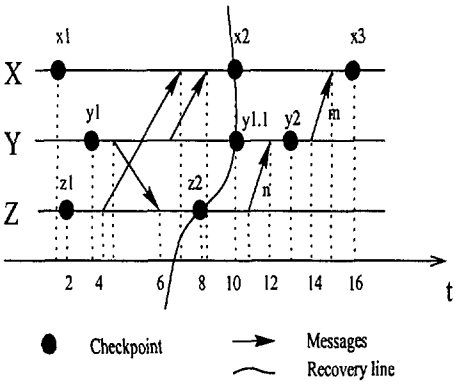


Figure 2: Recovery

same results again. This requires computations to be fully determined by received input messages. Restarting processes from an earlier checkpoint constitutes a backward error recovery strategy. An application process is said to be in "recovery" state if it has not yet reached the state at crash time. It is said to be "active" resp. "operational" if its computation proceeds beyond the crash state. Layer-2 software fault-tolerance strategies lead to increased application process availability, as well as increased message-space consistency.

A distributed application is said to be layer-3 software fault-tolerant if data kept in a file system are recoverable after a failure. If a process is restored from a checkpoint image, all files that were open at crash time should be accessible even if the process was restarted on another machine. Changes made to the files must be un-done prior to restarting from a checkpoint. Level-3 software fault-tolerance increases data consistency of applications and increases process availability as processes are able to migrate to another machine.

Layer-4 software fault-tolerance mechanisms are used if an application needs a very high availability. This is accomplished by replicating several copies of each application process on different nodes. When a process instance fails, identical output is available from another instance of this application process. Thus, the application continues to perform its computation apart from the time it takes to notice node failure, and to use results produced by another process instance. All it takes to implement layer-4 fault-tolerance is to synchronize replica behavior. Layer-4 software fault-tolerance increases process availability.

6 Recovery Design Alternatives on the Application Level

If a process of a distributed application has to be started from its last checkpoint, the question arises how to treat messages sent or received by the process since its last checkpoint. If e.g. process X in Figure 2 crashes, it may be restarted from checkpoint x_3 without affecting other processes belonging to the application. However, if process Y crashes at time $t = 14$ after sending message m, it will generate and re-send message m to process X when restarted from checkpoint y_2.

There are two basic alternatives for dealing with this problem. The first one, later referenced as Backward Backward Error Recovery (BBER), involves undoing all effects caused by a process in the time interval between its last checkpoint and the time of the crash. To undo the effects caused by a failed process on another active process, the failed process will be rolled back into an earlier state, and the other process will be restarted from a checkpoint too. Consider process Y failed after sending message m in the example. The BBER strategy would then require restarting process X from checkpoint x_2 to undo the effects of re-sending message m after Y is restarted from checkpoint y_2. The second alternative, later referenced as Backward Forward Error Recovery (BFER), ensures that a process restarted from a checkpoint executes the same instructions as on its initial run. However, effects affecting other processes are suppressed. Applied to the example, process Y would be restarted from checkpoint y_2. When restarted, Y will again generate and send m. Duplicate reception of m by X must then be suppressed by some external means.

The BBER may lead to a "domino effect", requiring the restart of other processes indirectly affected by a process abort. If process Z crashes after sending n, X, Y, and Z would need to be restarted from their respective checkpoints x_1, y_1, and z_1, as they

received some messages sent by Z after writing its latest checkpoint image. Within this context, messages n and m are called "orphan messages". Orphan messages may lead to a domino effect which possibly affects all application processes[5]. Applying the BBER strategy requires careful scheduling of checkpoints in order to avoid orphaned messages. In the best case there is no information flow at the time all application processes create a checkpoint image. This could be accomplished by scheduling process Y to write its checkpoint image $y_{1.1}$ at time $t = 10$. At this point in time, no unreceived messages are present in the system. If process Z would crash after sending n, restarting processes X and Y from checkpoints x_2 and $y_{1.1}$ would be sufficient to undo all changes made by process Z, which would then be restarted from checkpoint z_2. Checkpoints x_2, $y_{1.1}$ and z_2 are said to constitute a "recovery line", or "strongly consistent checkpoint". The drawback is that all processes must be considered when writing checkpoint images for every single application process. This requires synchronization among all processes in order to determine whether it is safe to write a checkpoint image.

Applying a BFER strategy alleviates the need for synchronization prior to taking checkpoints. Individual processes may write checkpoint images at any time. This requires keeping the message log in some entity surviving the process crash which is responsible for suppressing orphaned messages and replay of already received messages.

7 Combining Generative Communications and Software Fault-Tolerance

This section shows how software fault-tolerance mechanisms are added to programs based on the generative communication paradigm. Applying layer-3 methods yields an acceptable level of fault-tolerant execution for such applications. Application of layer-3 strategies to these distributed programs is then examined in greater detail. As all inter-process communication is done via the tuple space, there is already a system entity in place to keep the message log for each application process, unaffected by application process crashes. This lends itself to using the BFER strategy for process recovery.

On each machine where application processes are to be executed, a system service program is installed. Its task is to control and to restart application processes in the event of a machine crash. Thus layer-1 software fault-tolerance is reached. Layer-1 software fault-tolerance by itself is not sufficient for fault-tolerant execution of applications using the generative communication approach for inter-process communication as shared data may be left in an inconsistent state.

[5]There are more problems with BBER. An in-depth treatment is given in [MN94]

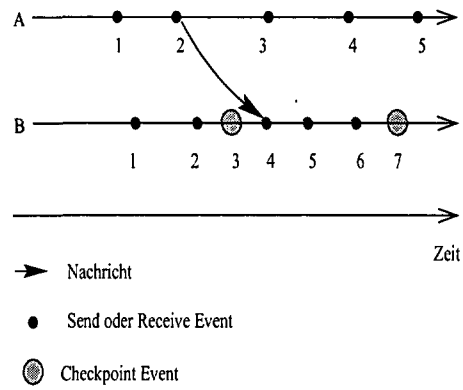


Figure 3: IPC using the Tuple Space

Layer-2 software fault-tolerance adds checkpoints and message logging to the layer-1 software fault-tolerance mechanisms. Applications implemented using the generative communication approach for IPC are already exchanging messages by adding and removing data tuples from a global tuple space, as depicted in Figure 3. Tuple space operations are kept in a per-process log. For each process, its checkpoint image freezes the state of the particular computation performed by the process. Messages sent or received as the checkpoint generation are commonly referred to as "events" and are also kept in the message log. Figure 4 shows the message log after exchanging messages in Figure 3. The call to `out()` in Process A is uniquely identified with event number 2. We use the BFER in order to re-integrate a crashed process into an application. It is sufficient to restart an application process from its latest checkpoint image and to supply messages from its message log. In particular, duplicate output messages may now be identified and are suppressed. When a process succeeded in taking a checkpoint image, events prior to the checkpoint event may be discarded from the message log. In Figure4, event 4 for process A would no longer be present in process B's message log as it is already incorporated into its checkpoint taken at event 7; process B would not receive this message again when restarted from this checkpoint. However, when process A is restarted from scratch after failing after event 5, the output message generated at event 2 must be prevented from reaching the tuple space, as this would create a duplicate tuple. Processes are said to be in recovery state when their communication is screened by a message log.

Distributed applications may need to access large amounts of data kept in files. If a machine fails and becomes unavailable, data kept on this machine is lost and may cause the entire application to fail. Layer-3 software fault-tolerance addresses this problem. It

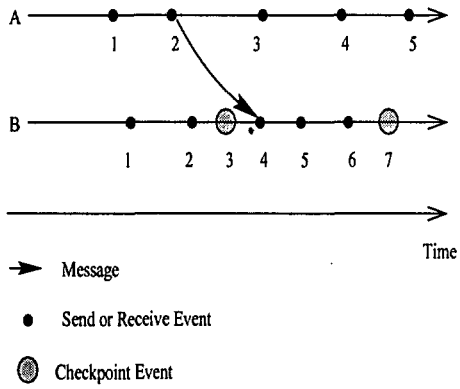


Figure 4: Message Logging

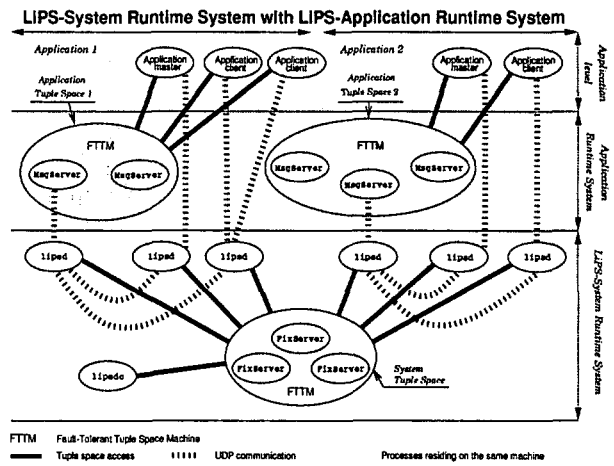


Figure 5: The different levels of the LiPS runtime systems

ensures that the file system environment[6] may be restored when encountering an error. Layer-3 software fault-tolerance may be implemented by replicating files accessed by application processes. If a process is to be restarted, all files required by the process have to be copied to its working directory prior to the process restart.

Normally, there is no need for application processes to be highly available. Applying layer-4 software fault-tolerance strategies is not necessary as applications are able to run to completion if layer-3 software fault-tolerance mechanisms are applied. Replicating individual application processes in order to gain increased availability would consume additional computing power which could be used by other application processes too.

8 The LiPS System

The main problem that arises in implementing a software fault-tolerant system for applications based on the generative communication paradigm deals with the question of how to make the Tuple space resilient to faults like machine crashes. Obviously, the solution to this problem is replication of the Tuple space among different machines. This approach is implemented very efficiently in the so-called Fault-Tolerant Tuple Space Machine [Set96, Set97] explained later in this section.

We distinguish two Fault-Tolerant Tuple Space Machines in the LiPS system. The first Fault-Tolerant Tuple Space Machine implements the System Tuple Space maintaining data about the system state e.g. which machine is idle. The second Fault-Tolerant Tuple Space Machine maintains the Application Tuple Space. Figure 5 on page 92 gives an overview.

There may exist several applications concurrently each using a private Fault-Tolerant Tuple Space Machine. The System Tuple Space is shared by all applications.

In this section, we first introduce the different components of the runtime systems of LiPS and their cooperation. A more detailed explanation is given in [SL97]. The design and implementation of the Fault-Tolerant Tuple Space Machine is explained next. A detailed description is given in [Set96].

8.1 The LiPS Runtime Systems

We distinguish two different runtime systems within LiPS . The system runtime system and the application runtime system. Both are based on a Fault-Tolerant Tuple Space Machine. The server processes of the Fault-Tolerant Tuple Space Machine for the system runtime system are called FixServer; those of the application runtime system's Fault-Tolerant Tuple Space Machine MessageServer. The relationship between the different runtime systems described above is depicted in Figure 5.

A designated server process called `lipsd()` resides on each machine participating in the LiPS system. The `lipsd()` processes update and retrieve information from the System Tuple Space. For example, node-state information, like load of a (the) machine can be read (updated) easily through Tuple space operations. `lipsd()` processes update their own node-state information in the System Tuple Space in fixed intervals. A machine crash can be detected if this information is not received in time. In this case possible errors due to lost data are repaired, and watchdog mechanisms will re-integrate the crashed machine "automagically" immediately after its recovery. A more detailed description of these mechanisms is given in [SF96].

Fault-tolerance on application level is implemented with a checkpointing and recovery mechanism in-

[6]The file system environment of an application consists of all files being accessed by an application process. Processes are expected to access files present in their working directories; in particular, no file may be open concurrently by several processes

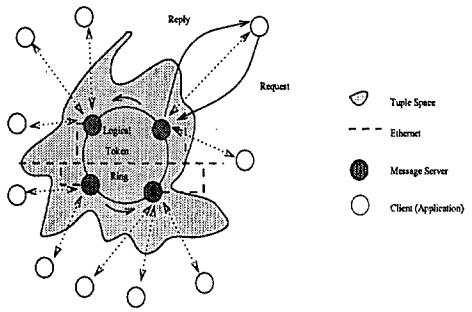


Figure 6: Processes of the Tuple Space Machine

tegrated into the Fault-Tolerant Tuple Space Machine. A checkpoint is correlated to the evaluation of an `eval()` operation; recovery is based on the re-execution of a failed `eval()` together with the replay of the message logging of the first execution of `eval()`. Message logging is provided via the Fault-Tolerant Tuple Space Machine.

8.2 The Fault-Tolerant Tuple Space Machine

The Fault-Tolerant Tuple Space Machine [Set96, Set97] replicates the content of the Tuple space among several machines. If a machine that a MessageServer (FixServer) resides on crashes, the data are still available on the replicas. An additionally started server process joining the Fault-Tolerant Tuple Space Machine will be initialized with the data of an old replica. This feature makes the Fault-Tolerant Tuple Space Machine N fault-tolerant. In the Fault-Tolerant Tuple Space Machine every tuple is tagged with a unique ID (Sequence Number) as a result of the protocol used to replicate data across the different machines. This unique ID is used to speed up replication of events among the different servers. The protocols used in the Fault-Tolerant Tuple Space Machine are based on those given in [ADM+93]. An in-depth description of the protocols used and their implementation is given in [Set96].

As depicted in Figure 6, the Tuple space is managed by several Message Server s residing on different machines. Message Server s must reside in the same broadcast domain. The broadcast facility is utilized to replicate messages very efficiently among the different servers. An additional token circulating among the servers schedules the permission to use the broadcast facility - avoiding Ethernet saturation due to collisions. The circulating token ships additional data enabling, among other things, flow control between the replicas. Additionally, each broadcast message (tuple) is tagged in sequence with a unique ID. This procedure establishes a linear order among the tuples of the Fault-Tolerant Tuple Space Machine and speeds up replication.[7]

As shown in Figure 6, an application process sends requests to the Message Server which is assigned to it. A request can either contain a tuple or a template. In the following, we first explain how Message Server s process a tuple, and second how templates are processed. As the Message Server s share the same broadcast domain, a Message Server is able to broadcast the tuple and hence replicate it on multiple Message Server s with only one physical operation. At any time only one Message Server may broadcast a tuple, namely the Message Server holding the token message. After a Message Server has finished broadcasting messages (tuples), it sends the token to the next Message Server . With respect to this token transfer, the Message Server s form a logical ring. Messages being broadcast are tagged with a unique sequence number. The sequence number of the last broadcast message of a Message Server is sent within the token. The next Message Server intending to broadcast knows the sequence number of the last broadcast message and continues the sequence, thereby establishing a total order on the messages broadcast. Within one token rotation several tuples may be broadcast by each Message Server .

If a Message Server receives a template, it first tries for a match on its local Tuple space . If no tuple matching the template is found, the Message Server notifies the requesting application process (NACK). Otherwise, if the Message Server finds a match, it must first synchronize with the other Message Server s. In order to notify the other Message Server s of the tuple access, it is sufficient to send the sequence number (4 bytes), the application process accessing the tuple and the event number in its message logging (4 bytes) as well as the type of access (1 byte) to identify the operation to the replicas. These items of access information now are added to the circulating token. The size of the token then determines the number of reading and extracting Tuple space operation s which may be replicated within one token rotation.

9 Summary

This paper addressed the basic design decisions made when building version 2.4 of the LiPS system for implementing fault-tolerant applications in networks of workstations.

As the application uses the tuple space for interprocess communication, applications are able to adapt smoothly to the workstation environment. Application processes may be recovered very efficiently using

[7] If a broadcast message was not received on a replica, this circumstance is easily obtained as there is a gap in the sequence of received messages. In this case, a retransmission could be requested immediately.

a recovery strategy based on resilient processes and resilient data. The advantage of this strategy is that application processes are independent both in the choice of when to take a checkpoint and when to recover from a checkpoint. This enables exhaustive usage of idle-time present in a workstation network as processes may be migrated to other idle machines in the event the processor they are running on becomes busy. The migration of a process can be based on the mechanisms used to guarantee fault-tolerance. This enables the system to rapidly and easily adapt to changes in machine usability such as those occurring during the daytime.

The above design buys efficiency from the implementation of a Fault-Tolerant Tuple Space Machine, replicating the content of the tuple space among different machines. The LiPS system distinguishes between two runtime systems both based on the Fault-Tolerant Tuple Space Machine. The first runtime system, the so-called system runtime system, provides the applications with software fault-tolerance of level 1 based on a watchdog mechanism. The second runtime system, the so-called application runtime system, provides the application with software fault-tolerance of level 3 based on checkpointing, message logging and the integration of files into the tuple space.

References

[ADM+93] Amir Y., Dolev P., Melliar-Smith P., Agarwal D., and Ciarfella P. Fast Message Ordering and Membership using a Logical Token-Passing Ring. In *13th International Conference on Distributed Computing Systems (ICDCS)*, number 13 in IEEE, pages 551–560, Pittsburgh, 5 1993.

[BDE94] Bakken D. E. *Supporting Fault-Tolerant Parallel Programming in Linda.* PhD thesis, The University of Arizona, 6 1994. Department of Computer Science.

[BS93] Bakken D. E. and Schlichting R.D. Supporting Fault-Tolerant Parallel Programming in Linda. Technical Report 93.18, Department of Computer Science, The University of Arizona, 6 1993.

[CFGK94] Carriero N., Freeman E., Gelernter D., and Kaminsky D. Adaptive Parallelism and Piranha. Technical Report YALEAU/DCS, Yale University Department of Computer Science, 2 1994.

[CKM92] Chiba S., Kato K., and Masuda T. Exploiting a weak concistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, 6 1992.

[GCCC85] Gelernter D., Carriero N., Chang S., and Chandran S. Parallel Programming in Linda. *IEEE Transactions on Computer*, 1985.

[Gel85] Gelernter D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GK92] Gelernter D. and Kaminsky D. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. *Sixth ACM International Conference on Supercomputing*, July 1992.

[KS90] Kambhatla S. Recovery with limited replay: Fault-tolerant processes in Linda. Technical Report CS/E 90-019, Department of Computer Science, The Oregon Graduate Institute, 9 1990.

[KS91] Kambhatla S. *Replication issues for a distributed and highly available Linda tuple-space.* Master thesis, Oregon Graduate Institute, Department of Computer Science, Beaverton, Oregon, 9 1991.

[LX89] Liskov B. and Xu A. A design for a fault-tolerant, distributed implementation of Linda. In *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, 6 1989.

[MN94] Mukesh Singhal and Niranjan Shivaratri. *Advanced Concepts in Operating Systems.* Series in Computer Science. Mc Graw Hill, 1994.

[PTHR93] Patterson L. I., Turner R. S., Hyatt R.M., and Reilly K. D. Construction of a fault-tolerant distributed tuple-space. In *Proceedings of the 1993 Symposium on Applied Computing.* ACM/SIGAPP, 2 1993.

[Set96] Setz T. *Integration von Mechanismen zur Unterstützung der Fehlertoleranz in* LiPS. PhD Thesis, Universität des Saarlandes, 2 1996. Fachbereich Informatik.

[Set97] Setz T. Design, Implementation and Performance of a Fault Tolerant Tuple Space Machine. In *Proceedings: ICPADS'97: 1997 International Conference on Parallel and Distributed Systems, December 10–13, 1997, Seoul, Korea.* IEEE, 12 1997.

[SF96] Setz T. and Fischer J. Software Fehlertoleranz vom Level Eins in LiPS. In Clemens H. Cap,

editor, *Proceedings of SIWORK'96, Workstations and their applications*, pages 102–112, Universität Zürich, Institut für Informatik, May 1996. vdf Hochschulverlag AG an der ETH Zürich.

[SL97] Setz T. and Liefke T. The LiPS Runtime Systems based on Fault-Tolerant Tuple Space Machines. In *Proceedings of the Workshop on Runtime Systems for Parallel Programming (RTSPP), 11th International Parallel Processing Symposium (IPPS'97), Geneva, Switzerland*, April 1997. Appeared as Technical Report, Vrije Universiteit Amsterdam, Faculteit der Wiskunde en Informatica, No. IR-417, februari 1997.

[SS83] Schlichting R.D. and Schneider F.B. Fail Stop Processors: An Approach to Designing Fault Tolerant Computing Systems. *ACM Transactions on Computing Systems*, 1(3):222–238, 3 1983.

[T.99] Setz T. Dynamic Load Adaption in LIPS. In *7th Euromicro Workshop on Parallel and Distributed Processing, Feb 3-5, 1999, Funchal, Portugal.* IEEE Computer Society Press, 1999.

[Xu 88] Xu A. *A Fault Tolerant Network Kernel for Linda.* Master thesis, MIT, Laboratory for Computer Science, Cambridge, 8 1988.

[YC83] Yennun H. and Chandra K. Software Implemented Fault Tolerance: Technologies and Experiences. In *Proc. of 23rd IEEE Conference on Fault Tolerant Computing Systems (FTCS)*, pages 2–9, 1983.

JavaPorts: An Environment to Facilitate Parallel Computing on a Heterogeneous Cluster of Workstations

Elias S. Manolakos and Demetris G. Galatopoullos
Electrical and Computer Engineering Department
Northeastern University, Boston, MA 02115, USA
Email: {elias, demetris}@cdsp.neu.edu

Keywords: parallel computing, clusters, Java, component programming

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 10, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

We present the JAVAPORTS *system, an environment and a set of tools that allows non-expert users to easily develop parallel and distributed Java applications targeting clusters of workstations. The* JAVAPORTS *system can automatically generate Java code templates for the* tasks *(software components) of an application, starting from a graph in which the user specifies how the tasks will be distributed to cluster nodes. Tasks have well defined* port *interfaces and may communicate by simply writing messages to their ports, without having to know the name and location of the destination task. This allows for task reusability while keeping the code for inter-task communication and coordination hidden. We demonstrate that coarse grain parallel programs developed using the* JAVAPORTS *system achieve good performance, even when using a 10Mbs shared Ethernet network of workstations.*

1 Introduction

The clusters of workstations provide a tremendous resource of spare CPU power at costs effectively much lower than that of conventional parallel machines. Clusters, with high-speed network connections, may exhibit performance comparable to that of supercomputers, primarily for coarse grain parallel applications. However, a drawback of such architectures has been their multi-platform characteristics. The introduction of the Java programming language brought to the table, among other valuable properties, the revolutionary concept of cross-platform programming. The JAVAPORTS system intends to utilize this capability and provide a user-friendly programming environment for the development of distributed concurrent applications on networks of heterogeneous workstations. It allows programmers who may not be very familiar with object-orientation and concurrency exploitation techniques, to easily develop, map and reconfigure parallel processing applications on heterogeneous clusters.

The JAVAPORTS environment, generates and updates plug-in software components which allow the distributed application to conform to the possibly changing nature of the cluster, while keeping the coordination of components hidden. Using a simple application configuration language, the programmer specifies a *Task Graph* and assigns tasks to machines. Tasks communicate via *ports*. A port is a software abstraction, which appears as a black-box to the developer who may use methods from a Java interface to write to, or read from, it every time s/he wants to transfer messages between two connected tasks.

The JAVAPORTS system will parse a specified Task Graph and generate Java *code templates*, one for each defined task, which may be used to complete the implementation of a specific application. The templates are well-structured, in that the definition and registration of ports is taken care of and only the local computational part of a task may need to be added by the programmer. When the developer modifies the assignment of tasks to machines, or adds/removes ports to/from a task, system tools will be called to update the affected templates while keeping the user-defined part of the code unaltered. This scheme allows for the reusability and incremental development of modular software components for parallel applications.

The JAVAPORTS tasks follow the Ideal Worker Ideal Manager (IWIM) model of *anonymous communications* [1, 2] i.e. a task may exchange messages with other tasks without knowing their identity, in contrast to the Targeted-Send/Receive (TSR) model. Each task is assumed to be an ideal worker who performs some job without knowing or caring about how the inputs it is using arrived at its ports, or where its outputs should be delivered.

There are several on-going research projects around the globe aiming at exploiting or extending the services of Java in order to provide frameworks for parallel applications development in different contexts. Due to lack of space, we can only mention a few here. Java

Parallel (Java//) [3] is a Java library that uses *reification* and *reflection* to support transparent remote objects and seamless distributed computing in both shared and distributed memory multiprocessors. The JavaParty package [4] also supports transparent object communication by employing a run-time manager responsible for contacting local managers and distributing objects declared as `remote`. (The JavaParty minimally modified the Java language in this respect). The JavaPP project [5] introduced the Communicating Java Threads that implement in Java the channels of Communicating Sequential Processes (CSP) model [6]. The JavaPP is a class library that supports synchronized message passing among concurrent processes in a real-time setting, where deadlines have to be respected. The JavaPP is targeting parallel applications for embedded systems, where the nondeterministic behavior of Java can be a serious drawback. Finally, the Javelin [7] project tries to expand the limits of clusters beyond local subnets, thus targeting large scale heterogeneous parallel computing. It is motivated from the popular idea of utilizing available CPU resources around the Internet for solving compute-intensive tasks. Javelin is composed of three major parts: the broker, the client and the host. Processes on any node may assume the role of a client or a host. A client may request available resources; a host is the process which may have and is willing to offer such resources to a client. Both host and client register their intentions with a broker; the broker takes care of any negotiations and performs the appropriate assignment of tasks. The user in the Javelin system interacts with a high-level API. A special purpose stack residing on the client side allows the user to push and pop tasks destined to be executed on remote nodes. Therefore the low-level details are hidden from the user. Message passing is a major bottleneck in Javelin, because the messages destined for remote nodes traverse the slow TCP or UDP stacks utilized by the Internet. However, for compute-intensive applications, such as parallel raytracing, Javelin has demonstrated good performance.

Our work conceptually differs from the above mentioned projects in that JAVAPORTS is an environment that forces the programmer, from early on in the application development stage, to configure his/her application as a collection of interconnected concurrent tasks with clearly marked boundaries which may exchange information using a simple communication mechanism. This software engineering approach is similar to the hierarchical methodologies used for designing complex digital systems with hardware description languages, such as VHDL [8]. The JAVAPORTS environment provides tools for capturing how the work partitions of an application should be distributed to the processors in a cluster. That means, the control over "what-goes-where" is given to the programmer. Once the task configuration has been decided, the system will automatically generate reusable software components, Java code templates, which may communicate asynchronously and exchange any type of objects using method calls to instances of the JAVAPORTS `port` class. This frees the programmer from having to worry about whether a communication is to a task running locally or to a remote machine, and from having to change her/his code every time s/he decides to modify the allocation of tasks to machines. The JAVAPORTS system attempts to provide, not only a simple and user friendly environment for the rapid prototyping of parallel applications on clusters, but also high quality parallel code in which hiding the coordination and communication details from the user does not come at the expense of adding a prohibitively large overhead.

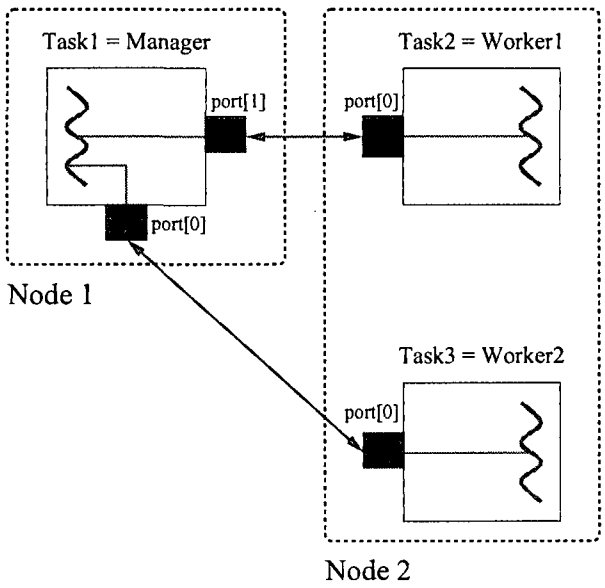
In the sequel, we introduce the various elements of the JAVAPORTS system. We first explain how a Task Graph can be captured, then discuss the structure of the generated Java code templates and how tasks can communicate using read/write methods of the `ports` interface. In section 3 we present some promising experimental results, and finally we summarize our finding and point to work in progress in section 4.

2 The elements of JAVAPORTS

2.1 The Task Graph

The developer utilizes the task graph in order to map the work partitions of the application onto the nodes of the cluster. An example of a task graph and its corresponding configuration file is shown in Figure 1, where the solid boxes denote tasks and the dotted boxes, cluster nodes. There are three user-defined tasks in this example, one (called the Manager) allocated on Node1 and another two (called Worker1 and Worker2) allocated on Node2. The Manager uses a pair of ports to communicate with the two remote workers.

When the user compiles the configuration file, the JAVAPORTS system will extract the information needed to create, or update, the task code templates and scripts that are necessary to launch the distributed application. If at a later time the programmer modifies the configuration file, s/he may recompile it in order to update these templates and scripts. This recompilation will not affect at all the user-specified part of the code template of each task. The JAVAPORTS system will perform all the necessary changes to be able to execute the same application correctly on the new cluster setup, even in the case that the user decides to assign all the tasks to the same single node. The syntax of the configuration language and the JAVAPORTS*Application Configuration Tool* (JACT) em-




```
begin configuration
  begin definitions
    define      machine Node1= "corea.cdsp.neu.edu"
    define      machine Node2= "walker.cdsp.neu.edu"
    define      task Task1= "Manager"
    define      task Task2= "Worker1"
    define      task Task3= "Worker2"
  end definitions
  begin allocations
    allocate    Task1    Node1
    allocate    Task2    Node2
    allocate    Task3    Node2
  end allocations
  begin connections
    connect     Task1.port[0]    Task3.port[0]
    connect     Task1.port[1]    Task2.port[0]
  end connections
end configuration
```

Figure 1: A Task Graph and the corresponding JAVAPORTS configuration file.

ployed for the development of modular, reusable software components, are discussed in detail in [9].

2.2 Task Code Templates

The templates generated for Task1 (Manager) and Task3 (Worker2) of Figure 1 are outlined in Figure 2. The shaded parts are created by the JAVAPORTS system and what comes in between them corresponds to user-added code.

The `configuration()` method of the `Init` class instance is responsible for instantiating the port objects for each task. The `Init` class is a JAVAPORTS system class and is not visible to the programmer. The Init object will perform the RMI [10] port object registrations as well as the lookups of the corresponding peer ports. These time-consuming operations are performed transparently to the user and only once during the initialization phase of each task.

The task templates are prototype classes which implement runnable objects. Each template contains a `main()` method which is responsible for spawning a thread to run the task object. (The `main()` methods are depicted at the bottom shaded areas in Figure 2). The user code implementing the domain-specific operations of a task may be entered inside the `run()` method of the task template.

A Java interface, containing methods that can be applied to port objects, is also provided. The user will embed such method calls in his/her code every time a task needs to exchange information with another task. The read and write port operations are asynchronous and they are implemented by JAVAPORTS system classes. The user program will issue reads and writes to the local ports of the task. In succession, the JAVAPORTS system will make calls to the RMI runtime environment which will transparently transfer the messages to the remote objects. The user does not need to be concerned with where the destination object is located or how the messages will reach it.

In single compute node concurrent applications, the communication implementation remains unaltered. Peer ports will be registered and looked up in a manner similar to that of the distributed case. Nevertheless, since these operations are performed during the initialization phase of a task, no considerable overhead is added to the computational part of the application. We are currently investigating methods that may detect and exploit the locality of tasks to reduce the overhead incurred during message exchanges among local ports.

2.3 The `Ports` Java Interface

An interface in the Java programming language is an abstract class consisting only of public abstract methods and static fields. A class is used to implement all the methods included in its interface. The concept of an interface was introduced in Java to compensate for the lack of multiple class inheritance capabilities. In the context of JAVAPORTS, an interface will supply to the user all the permitted port-to-port communication operations while keeping their implementation hidden. This port interface currently includes the following methods:

`-AsyncWrite(Msg, MsgKey)`: It will write the `Msg` message object and its personalization key `MsgKey` (an integer) to a port. The message key can be used by the receiving task in order to identify the specific message among a list of messages received at its port. So although the receiving task does not need to know the

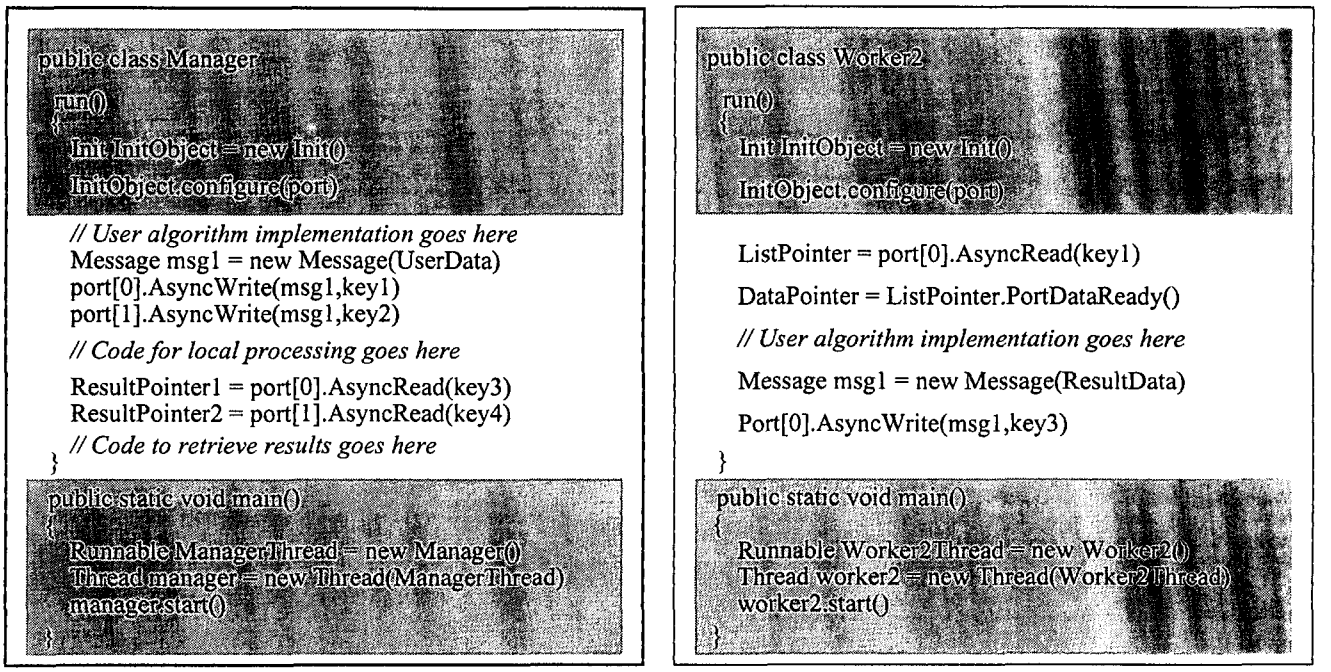


Figure 2: *Left:* Template for the Manager task; *Right:* Template forthe Worker2 task.

name of the sending task, it should know the key number for each message it expects at a port. This method will execute concurrently with the task that calls it (non-blocking write).

`-AsyncRead(MsgKey)`: This method is issued when a task wants to access a message object arriving at a port. It will return a handle to an object element of a linked list. This list is used by the port in order to deposit the arriving messages. The task may use this handle at any point to access the data part of a message with key value `MsgKey` that has arrived at the port.

`-PortDataReady()`: This accessor method will return a handle to the object in which the data part of the incoming message was stored. The task will use this reference in order to retrieve the data. The details of this operation are described in the next section.

The messages exchanged between tasks are objects of the class `Message`. This is a user-defined class and it extends the `Object` class. Therefore, it adds the capability of encapsulating any possible type of information in a message. Furthermore, the user-defined variable `MsgKey` is used in order to guarantee correct delivery of messages between tasks. It is the responsibility of the programmer to ensure that unique matching keys are used on the messages exchanged among two communicating tasks. In case that the programmer writes two messages with the same key to a port, the message that arrives second at the destination port overrides the one that arrived earlier. Therefore, it is imperative that the programmer treats the `MsgKey` variable with caution.

In the example of Figure 2, the Manager task writes a message to two ports connecting it to two almost identical worker tasks. Each worker reads its port and after performing some computation using the received message, it writes back the results that will be read by the Manager. The two worker templates may differ only in the local computation part and in the keys used in the read and write method calls. This shows how easily templates can be reused when the communication pattern remains the same.

The user program templates are defined as part of the JAVAPORTS system class package. Therefore, after the application implementation is completed, the templates may be compiled and executed along with the system classes. For cluster of workstations where the Network File System (NFS) is available, the programmer may execute any task from any cluster node without the need of distributing the template executables from node to node. In clusters where NFS is not available, specialized scripts are automatically created by the JAVAPORTS system in order to aid the user in distributing the executables to their correct destination nodes [9].

2.4 The Communication Protocol

In this section we outline the operation of the JAVAPORTS system during a message communication operation between two distributed tasks. A pictorial description is provided in Figure 3, where square boxes depict objects and the label inside a box refers to the class from which the object was instantiated. Labels of

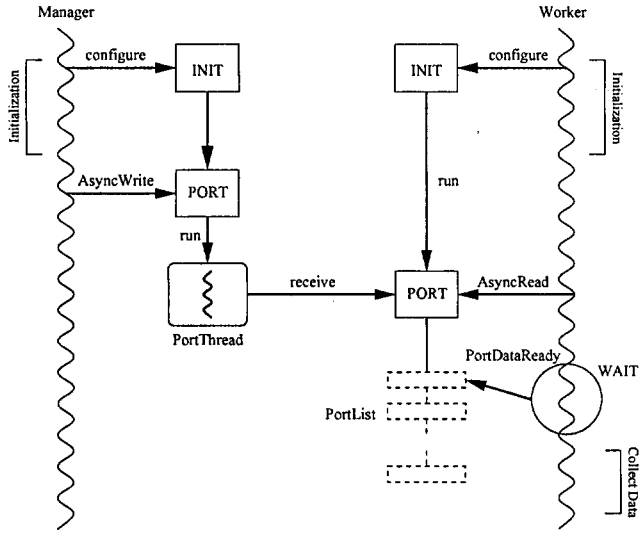


Figure 3: Port-to-port communication protocol using JAVAPORTS.

arrows denote calls to methods of the object receiving the call.

After each task thread has started, it makes a call to the `configure` method of the `Init` object in order to create and initialize all its ports and connect each one of them to its peer remote port. The `configure` method will perform all the appropriate calls to the RMI environment in order to register local, and lookup remote, ports. Subsequently, the method will return to the calling thread a (zero-based) array of port handles. Each port maintains a dynamically allocated linked list of objects (denoted by dashed boxes in Figure 3) that will be used to store incoming messages arriving at the port.

The direction of information flow does not affect the way communication is realized, therefore we assume, without loss of generality, that the manager task wants to send a message to a worker task directly connected to it. To do so, the manager task thread will initially invoke the `AsyncWrite` method on its local port. The parameters passed to this method are the message object and its associated personalization key (an integer). The remote worker task will use this key when it needs to retrieve the data encapsulated in the message. The `AsyncWrite` will spawn a `PortThread`, an active object which will be responsible for implementing the actual message transfer to the remote port, while the main manager task continues with the execution of its local computation (non-blocking send). The main function of the `PortThread` object is to make a remote call to the `receive` method of the worker task port. This method uses the RMI serialization mechanism in order to perform the actual transfer of the message object to the remote port.

At the receiver's end, after initialization, the worker thread makes a call to the `AsyncRead` method of its corresponding local port, using the key for the expected incoming message as a parameter. The `AsyncRead` method simply returns a handle to an object of the port list where the incoming message, with a matching key value, will be deposited upon arrival. At the appropriate point in time, i.e. when the worker task really needs the data in this message, it may use the handle to perform a call to the `PortDataReady` accessor method of this list object. This method call will block the worker task until a message with this key value has arrived at the port (wait-by-necessity). When the `PortThread` object of the manager makes the call to the `receive` method of the port object of the worker, the message will be delivered. Furthermore, this port will use the key value to pass the message to the appropriate object in its port list. This list object will be responsible to notify the (possibly waiting) worker task thread that a message has arrived. Upon notification, the worker task will exit the wait state and check if the message that arrived is the expected one. If so, it will retrieve the data, otherwise, it will re-enter the wait state.

The same sequence of operations may be used in the reverse order, if it is desired to send a message from the worker to the manager task using the same pair of peer ports. So communication is non-blocking and bidirectional.

3 Experimental Results

In this section we outline some experiments conducted using the JAVAPORTS system and discuss the performance results obtained. The (square) matrix-vector multiplication problem was solved in a variety of node configurations, exactly as it was done previously in [3] and [11].

We experimented with 1-, 2- and 3-node multiplication scenarios. In multi-node configurations, as many tasks as nodes (machines) were used. That means, a manager task was allocated to a node (called the master) and in addition one worker task was allocated per node to one (two) more node(s). The set of rows of the multiplicand matrix needed by each node were made available to it before the commencement of the experiment. The manager task (running at the master node) starts first by sending the multiplier vector to all remote workers; then it performs its part of the matrix-vector multiplication and waits (as needed) for the remote workers to send back their results, so that it can finally construct the overall result vector. The time it takes for the manager to distribute the multiplier vector to all the workers as well as the time the remote workers need to compute and send their results back to the manager was measured and taken into consideration in the timing analysis.

In our experiments, a size 1000×1000 matrix was multiplied by a 1000×1 column vector. The method `CurrentTimeMillis` of the `System` class, which re-

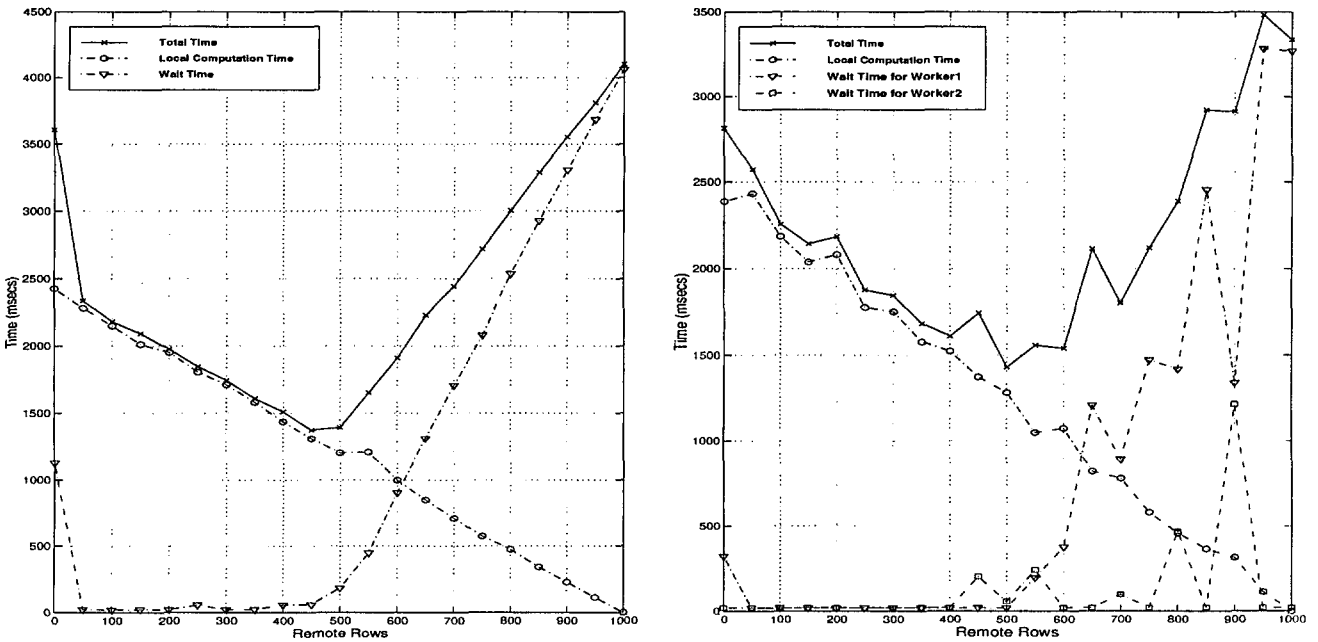


Figure 4: Timings for the Manager task. *Left:* 2-node scenario; *Right:* 3-node scenario.

turns the current time in milliseconds, was used to gather measurements. We acquired the time at the beginning and at the end of each targeted period and the elapsed time was determined by calculating the difference of the two measurements. For all the scenarios investigated the total elapsed times were decomposed into their sequential components that are also plotted in Figure 4.

The JAVAPORTS system was configured on a set of Sun Microsystems Workstations running Solaris v. 2.5.1, connected to the same subnet (10Mbps standard Ethernet). The type of machines used was either Sun Sparc-4 or UltraSparc-1. In the 2-node scenario, both nodes, the one running the manager task and the other running the worker task, were Sparc-4s. In the 3-node scenario, the node running the manager task was an UltraSparc-1 and the other two nodes, each one running a worker task, were Sparc-4s. (Therefore, the machines that run worker tasks were always chosen to possess identical hardware characteristics).

In the 2-node scenario, the number of rows of the multiplicand matrix that resided on each node was varied from 0 to 1000. In the left panel of Figure 4 we show the total time measured for the manager task as a function of the number of rows allocated to the remote worker task. (For example, 300 "remote rows" in this figure means that the remote worker task computed the last 300 elements of the product vector and the manager task computed the first 700 elements). The dashed-dotted curves depict (i) the local computation time of the manager and (ii) the time the manager had to wait, after its local computation is completed, for the worker results to become available. We notice that the minimum total time is observed when the rows of the matrix are almost evenly split among the two nodes. Furthermore, as the number of rows processed by the remote worker increases, (i) the manager's local computation time decreases, and (ii) the time the manager has to subsequently keep waiting for worker results to arrive increases, as expected.

In the 3-node scenario, the multiplicand matrix was split into three not necessarily equal parts. The manager task (in the master node) is first assigned a number of rows ranging from 0 to 1000. The remaining rows are now split among the two worker nodes. The right panel of Figure 4 summarizes the timings obtained for the manager task. For each number of remote rows processed collectively by the two remote workers, we report the minimum total time observed over several attempted work decompositions among the two worker nodes. Similarly to the 2-node scenario, the minimum total time is obtained when the work performed locally by the manager matches the work performed collectively by the two remote workers. The time that the manager has to wait (upon completion of its local computation) for each worker to return results is different, because the wait period for Worker2 results starts after the wait period for Worker1 results is completed.

The timing results for all experiments are presented together for visual comparison purposes in the left panel of Figure 5. The horizontal line corresponds to the time one workstation (Sparc-4) using a single thread pure Java program needs to perform a 1000×1000 matrix-vector multiplication and is included for reference. An 1-node (Sparc-4) concurrent

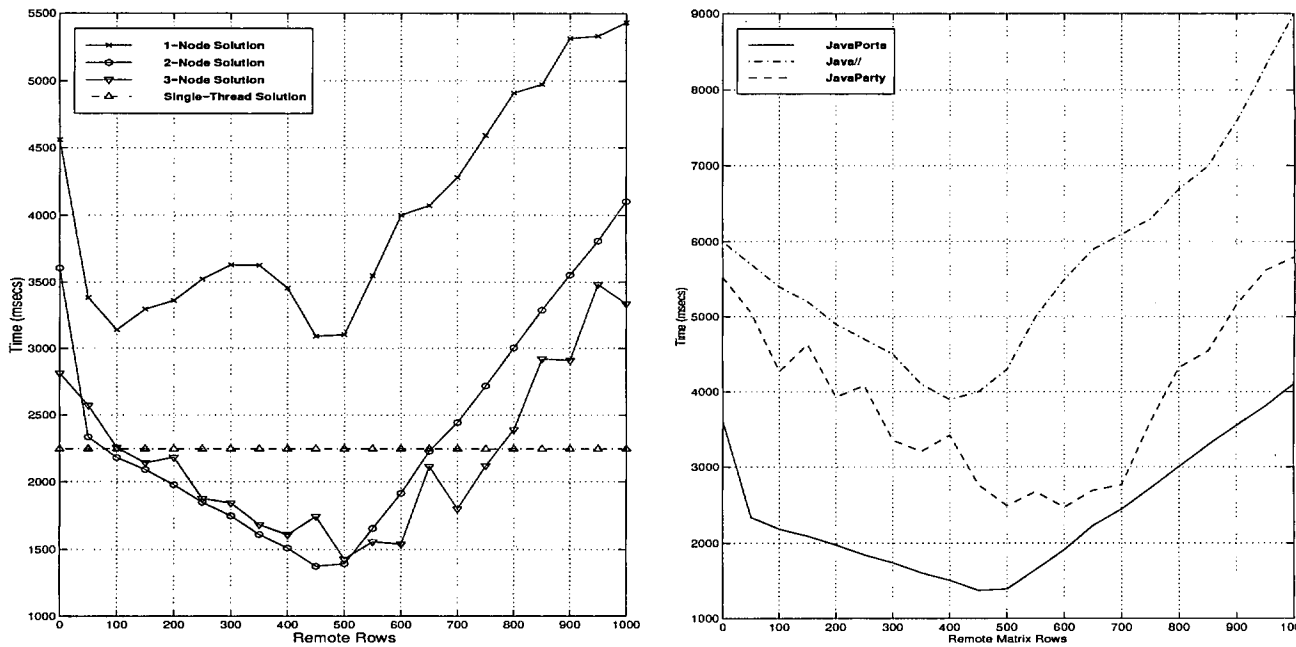


Figure 5: *Left:* Comparison of the 1-, 2- and 3-node scenarios. *Right:* Comparison of JAVAPORTS, Java// and JavaParty, for the 2-node scenario.

JAVAPORTS implementation, using one manager and one worker task, was also included to demonstrate the advantage obtained by task distribution when a second compute node is introduced to run the worker task. The difference of these two curves quantifies the overhead of having two JAVAPORTS concurrent tasks trying to solve the problem in a single compute node. The 2- and 3-node scenarios outperform the single-thread pure Java solution for a wide range of rows processed remotely. Moreover, the 3-node solution is consistently faster than the 2-node one, and their performance difference becomes more profound as the number of rows allocated to the remote workers increases.

The JAVAPORTS system performance was also compared to that of two other packages, namely the JavaParty [4] and the Java// [3], using the same 1000×1000 matrix-vector multiplication problem running on 2-nodes. To obtain the JavaParty timings we wrote our own simple Java program. Following the guidelines of the JavaParty group the worker task was declared as a remote class, thus allowing the system to migrate the worker task object to the remote node. On the contrary, the Java// results were not produced by us, but taken from [3] where the authors used two UltraSparc Sun workstations to perform the exact same matrix-vector experiment. As it can be seen from the plot in Figure 5, the JAVAPORTS system exhibited the best performance. Note that the JAVAPORTS and JavaParty experiments were run on two Sparc-4s that are less powerful than the UltraSparcs used for the Java// experiments.

We have also performed a detailed benchmark analysis of the standard Ethernet network which interconnects our cluster of workstations. To do so we performed the commonly used "ping-pong" experiment between two workstations. The manager task, running on the first node, initiates the communications by sending a message to the worker task running on a different node. Once the message is sent, the manager task performs an `AsyncRead` on its local port and waits. The worker task receives the message on its corresponding local port and it immediately writes it back to the port. Once the manager task receives the message, it records the roundtrip delay and it repeats the same action. For each message size used, the experiment was repeated 500 times and the average roundtrip delay was calculated. As a message we always used an array of words (64-bit double numbers). In each experiment we varried the array size. We first tried small sizes (1, 10, 100, 1000 words). Then, for larger arrays of one thousand (1000) elements and higher, we incremented the size by one thousand (1000), until the array was ten thousand (10000) words long.

The average roundtrip delay for each experiment as a function of the message size is shown in the left panel of Figure 6. The straight line corresponds to the least squares polynomial fit over the measured data points (for array sizes larger than 1000 words). The coefficients of this first degree polynomial can provide estimates for the message setup time and the per-word transfer time experienced when two JAVAPORTS tasks residing on separate address spaces communicate. The setup time is estimated to be in the order of 14.43msec

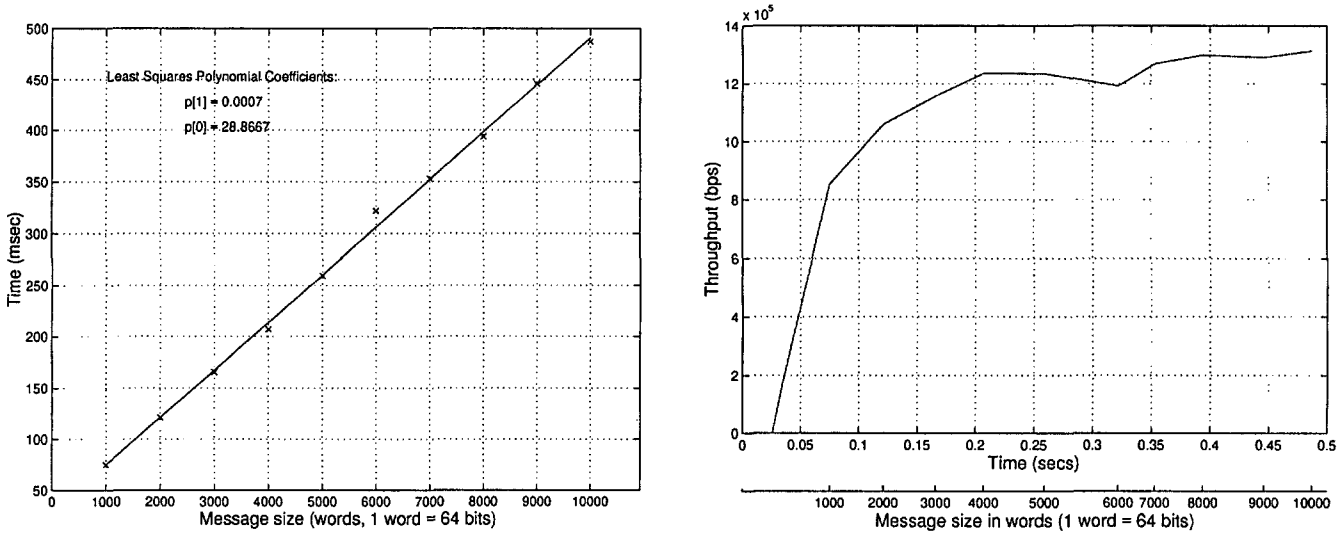


Figure 6: *Left:* Round trip delay vs. message size in Ethernet; *Right:* Ethernet Signature Graph.

(per message) and the transfer time 44.8μsec per word.

The *network signature* graph [12] is also provided in the right panel of Figure 6. Such a graph shows the transfer speed versus the elapsed time for each message size. As it can be observed, the Ethernet network in our experiments reaches saturation at messages sizes of 4000 words, giving maximum throughput in the order of 1.3 Mbps. Considering that the maximum throughput for transfering large messages over Ethernet is in the range of 7Mbits [12], what was achieved is respectful performance. The underutilization of the network can be contributed to several factors. First, the JAVAPORTS system makes use of RMI, whose message serialization and de-serialization process is known to slow down message passing. Secondly, the experiments were conducted on a local area network (LAN) at a time that its bandwidth could be possibly shared by other users. Finally, a Network File System (NFS) service was running on the LAN during the experiments, which may cause some delays since the workstations need to constantly contact their peers for file reconciliations and updates.

4 Conclusions

We have presented an overview of the JAVAPORTS system, an environment for flexible and modular concurrent programming, on a cluster of workstations. The design encourages reusability by enabling the developer to build parallel application using Java in easy modular steps. The components which comprise such designs may be manipulated through the system and without modifying any existing user code they may be used to reconfigure and run a parallel application in a completely new cluster setup. The user-defined message object may be altered to customize the needs of transferring any type of information across the network. By giving the developer such capabilities, the JAVAPORTS environment may be customized for specific client-server applications where a large variety of services may be available.

Each task in the JAVAPORTS system has its own separate address space. The system was not intended to support distributed shared memory (DSM) that would permit sharing of objects among tasks at run-time. However, a specific task may spawn as many threads as the programmer desires and these threads may share data residing in the task's address space, as in any concurrent Java program. Message communication in JAVAPORTS is anonymous, therefore one can "disconnect" a task from a port and "connect" another one to it and still the programmer of the task at the other end of the connection does not need to be informed about it, or change the communication code of this task in any way. This plug-in capability, allows to share task code templates among similar applications and accelerate application development. Users of the system do not have to be experts in parallel computing, but rather in the specific task they want to implement. With the JAVAPORTS system, one can take software components generated by different field experts and build easily a distributed application using them.

The performance observed in small scale benchmark applications (also tried by other groups) is very promising and shows that the JAVAPORTS system strikes a good balance between added functionality and performance. The inter-task communication protocol is quite simple and has built into it latency hidding capabilities. We are not so much concerned about performance issues, since we expect it to improve drastically with the rapid advancements in JVM and RMI implementations. We are currently designing larger scale distributed applications using a variety of task

graph configurations (star, pipeline, mesh etc.). We are also planning to add to the system dynamic port creation and task migration capabilities in the future.

References

[1] F. Arbab. The IWIM model for coordination of concurrent activities. *Coordination '96, Lecture Notes on Computer Science*, vol. 1061, April 1996.

[2] F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaas. Reusability of Coordination Programs. Technical Report CS-TR9621, Centrum voor Wiskunde en Informatica, The Netherlands, 1996.

[3] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, Nov 1998.

[4] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[5] G. Hilderink, J. Broenink, and A. Bakkers. A new Java Thread model for concurrent programming of real-time systems. *Real-Time Magazine*, pages 30–35, January 1998.

[6] C.A.R Hoare. Communicating Sequential processes. *Communications of the ACM*, pages 666–677, August 1978.

[7] B.O. Christiansen et al. Javelin: Internet Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, November 1997.

[8] Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems.* McGraw Hill, 1998.

[9] D.G. Galatopoullos and E. S. Manolakos. Developing parallel applications using the JavaPorts environment. In *International Parallel Processing Symposium (IPPS/SPDP-99)*, April 1999, to appear.

[10] Remote Method Invocation Specification. Sun Microsystems, Inc., 1996/1997.

[11] R.R. Raje, J.I. William, and M. Boyles. An asynchronous remote method invocation (armi) mechanism for Java. *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[12] Q.O. Snell, A.Mikler, and J.L. Gustafson. Netpipe: A Network Protocol Independent Performace Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

Structured Performability Analysis of Parallel Applications

John P. Dougherty
Department of Mathematics and Computer Science
Haverford College, Haverford, Pennsylvania 19041-1392 USA
Email: jdougher@haverford.edu

Keywords: performance evaluation, clusters, NOWs, fault tolerance, Synergy

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 14, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

Structured Performability Analysis (SPA) is a unified performance and dependability evaluation methodology for practical, large scale parallel applications. This evaluation can be used to guide the developer before and during design and implementation. SPA involves a systematic set of steps to decompose a large application for top-down as well as bottom-up analysis. SPA is scalable, supports variable-precision modeling, and is trackable. A brief overview of SPA is presented. This is followed by preliminary results correlating observed performance from experiments with expected performance from the SPA model, as well as dependability and performanbility projections.

1 Introduction

The primary purpose of parallel computing is to increase realized computation speed. Significant, but often not considered as important, is dependability [7]. Dependability is a serious issue for parallel and distributed applications because increasing the number of processors to increase performance can decrease overall dependability, especially for large-grained, loosely-coupled parallel systems. Fault tolerant techniques exist for increasing dependability at the cost of performance [11]. An analytical measure of the relationship between dependability and performance does not yet exist for parallel applications [10].

A parallel application typically contains multiple parallel processing topologies. These topologies were classified by Flynn in [6]. Each topology has its own characteristic relationship between performance and dependability. Any technique to gauge the performance and/or the dependability of the entire system must be able to gauge the contribution of every component relative to its cost.

An open question is, "How can the tradeoff between performance and dependability be determined to identify optimal delivered performance for parallel applications?" The purpose of this manuscript is to present a systematic analytical approach for investigation of this question, and to provide a unified metric of performance and dependability for parallel and distributed applications.

2 Background

Parallel processing is driven by the increasing need for computing speeds mandated by "Grand Challenge"-type applications, and by the realization that we are fast approaching the performance limits of present semi-conductor materials [13, 17].

Performance evaluation techniques for parallel applications have concentrated on quantifying realized processing rates and execution times, while disregarding dependability consequences.

While there are no universally accepted methods for performance calibration of parallel applications, there are accepted models for dependability. Typically, dependability models capture the possible states a system can reach, and then partition states into two categories: available or unavailable. Recently, dependability models have been retrofitted to include performance issues as rewards [1, 16].

As parallel applications grow in demand, they will also grow in size and complexity. Metrics to measure performance and dependability will need to scale so that they are useful for current as well as future applications. It is becoming increasingly difficult to study large parallel systems and applications using standard, unstructured analysis [8]. The problem addressed is how to gauge delivered performance (*i.e.*, performance considering failure) for parallel applications using a structured analytical method.

Performability, or delivered performance, refers to the solving of state-space type availability models that have been augmented to include speed characteristics in the form of rewards. The two most accepted performability approaches are Markov reward models (MRMs) [16] and stochastic reward nets (SRNs) [1].

Performability analysis is difficult for sequential applications; analysis of parallel applications is more complicated due to multiple cooperating processes, multiple processors, interprocessor communication (IPC), and storage issues of the parallel environment.

A number of issues are relevant to this research: i.) there is no universally accepted model of parallel processing [15]; ii.) models for performance analysis do not consider the runtime mapping of application to environment; iii.) the performance function $F(t)$ for performability analysis often represents the probability that a task will complete by time t [12]; and iv.) analysis of parallel applications needs to be conducted in steady-state.

Performance and dependability evaluation methods need to be scalable. Programming paradigms are emerging which increase application size and complexity. While these improvements are beneficial, it becomes more problematic to develop general-purpose techniques for parallel performance and dependability analysis.

The primary goal of this research project is to develop a unified performance and dependability model for parallel applications. This model is directly related to practical program execution by including processing rate and execution time for performance analysis. Dependability is quantified as availability, and is expressed as a probability function.

3 Methodology

Structured Performability Analysis (SPA) uses a three phase methodology to reach the goals of variable-precision and scalability. This methodology is depicted in Figure 1. The first phase involves a top-down Coarse-To-Fine (CTF) Application Decomposition. This results in a *Structured CTF (SCTF) graph.* The SCTF graph is then translated into its equivalent *Canonical Decomposition Tree (CDT).* The CDT sets the stage for later performability synthesis.

The second phase of SPA is *Microanalysis* that produces estimated performance and dependability functions for each of the application components. These functions form a performability profile for a given component.

The last phase involves a bottom-up synthesis termed *Macroanalysis.* Macroanalysis utilizes component performability profiles, performance and dependability template rules (*i.e.*, sequential, pipeline, SIMD, MIMD), and the application-dependent CDT to generate a performability profile of the entire application. The result is a combined measure of performance and dependability that considers both application-dependent and processing environment-dependent issues.

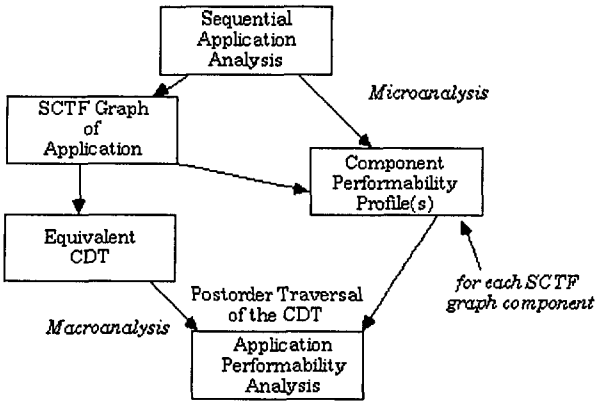


Figure 1: **Overview of SPA.**

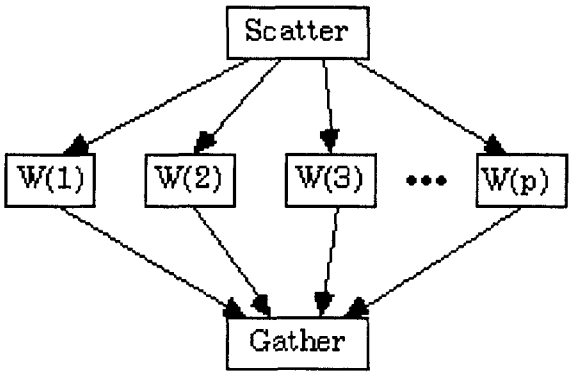


Figure 2: **SCTF Graph for SAG.**

4 A Brief Example

Consider a standard Scatter-And-Gather (SAG) parallel application. The sequence of operations in this application includes scattering the work, computing the work assigned, and gathering the results. Typically, a sequential application is used for speed comparisons.

Figure 2 is an example of an SCTF graph for an SAG application. The SCTF graph shows the typical scatter and gather operations and system configuration.

This SCTF graph is next transformed into a corresponding CDT as presented in Figure 3. Note that the vertices of the original SCTF graph in Figure 2 appear as leaves in the CDT of Figure 3. The internal nodes capture the structure of the original application. In this example, the SAG application is decomposed into a parallel (SIMD) structure nested inside a sequential (SEQ) structure. Application components are mapped onto the processor cluster as depicted in Figure 4.

Microanalysis involves developing a performability profile for each distinct leaf in the CDT of Figure 3. A

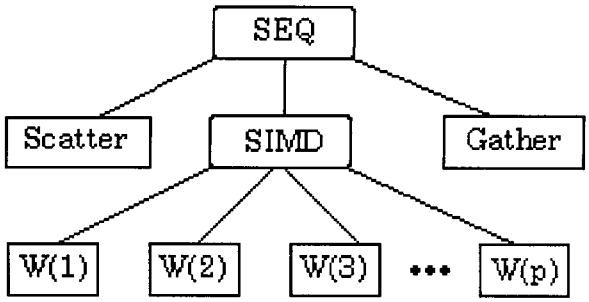


Figure 3: **CDT for SAG.**

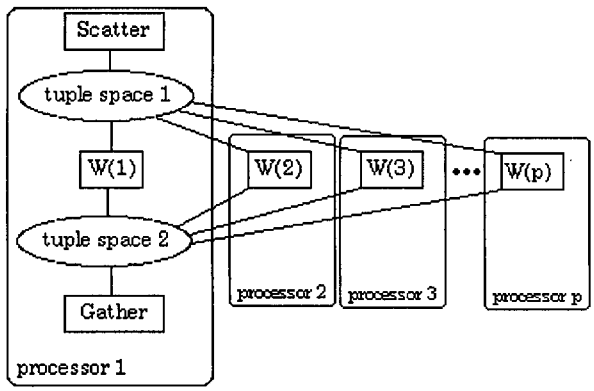


Figure 4: **Resource Allocation for SAG.**

performability profile consists of functions for execution time, processing rate, availability, and performability of a specific component. For the SAG example, there are three distinct types of leaves; namely, scatter, gather, and worker(i). The expected execution time of a worker process is typically the time needed by the sequential application divided by the number of replicated workers. Other functions are developed for each CDT leaf using template rules [5].

Macroanalysis involves traversing the CDT in postorder using template rules to derive functions for the internal CDT nodes. The resulting profile at the root of the CDT is the SPA profile for the entire application. Macroanalysis is detailed in [5]

The primary contribution of this manuscript is a scalable analytical method for assessing the combined performance and dependability of parallel and distributed applications. While it is generally known that certain properties of a parallel application are responsible for its performance and dependability characteristics, the exact details of these properties (and their interdependencies toward the aggregate contribution to overall system performance and dependability) are

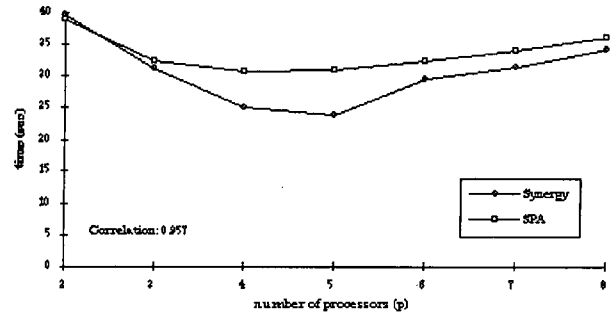


Figure 5: **Peak Execution Times.**

generally not known.

5 Results

As a case study, sequential and parallel matrix multiplication applications were implemented for a cluster of DEC Alpha workstations interconnected via Ethernet. The parallel application employed SAG, and execution times were collected after tuning to minimize synchronization overhead [4]. Process coordination was implemented using Synergy [14]. Calibration runs were used to determine such parameters as average processor rate, IPC rate for the SPA model. Execution time functions, both observed using Synergy and expected using SPA, are depicted in Figure 5.

The correlation near unity between observations and expectations support the use of SPA as a tool to evaluate application changes (*e.g.*, impact of increasing problem size), environment enhancements (*e.g.*, impact of a faster interconnection network), and/or optimization scenarios (*e.g.*, processor count where time is minimized).

SPA also produces performability projections, which are given in Figure 6. Peak performance does not consider the impact of partial failure. Weak delivered performance results when no fault tolerance is implemented, implying that partial failure results in total application failure. Strong delivered performance occurs when the replicated workers are designed to support failover [9].

SPA can be utilized to augment the previous analysis to consider dependability issues as part of application performance. Benefits of fault tolerance schemes can then be weighed against expected costs to performance.

6 Conclusions

The goals of the research consist of i.) a unified, structured model of speed and availability; ii.) a scalable approach; iii.) variable-precision; and iv.) a trackable approach. SPA achieves unified performance and

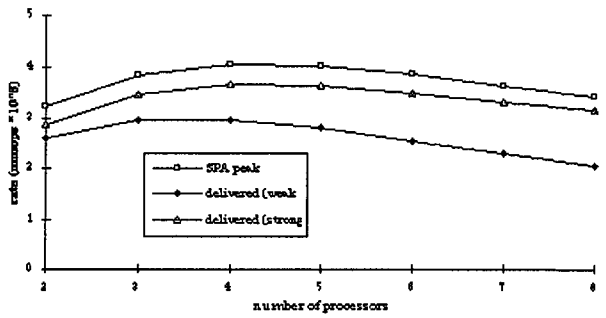


Figure 6: **Peak and Delivered Rates.**

dependability through the delivered performance functions derived as part of the performability profile for an application. SCTF graphs are used as performance graphs to derive peak and delivered processing rate, as well as availability. Derived performability provides a method for measuring speed considering partial failure.

SPA provides a scalable methodology for studying the speed of parallel applications. Microanalysis parameters capture the essential speed-critical factors of both the application and the architecture. These parameters can scale independently. SPA can be used for a variety of applications and architectures available now and in the future.

The SPA model supports variable precision to balance simplicity with detail, and provide a means for increasing or reducing detail as warranted. This balance is the responsibility of the application programmer.

SPA supports a trackable methodology because the developer can perform top-down or bottom-up analysis. The CDT captures the functional dependencies among the application components. Macroanalysis provides a top-down profile of application speed and availability. Changes made at the component level can be traced to their impact on overall application speed and availability. This bottom-up analysis permits incremental changes in the application to be studied early in the design process, facilitating rapid prototyping.

Future research includes enhancements to Timing Models to quantify communication rate simply and accurately [3], as well as non-probabilistic measures for dependability such as fuzzy metrics. Various applications and various platforms are currently being investigated to ensure that SPA is not bound only to NOWs and clusters.

SPA provides a way to investigate the internal structures of a large parallel application for both performance and dependability contributions. The model identifies the thresholds of key performability parameters. This makes optimization possible prior to implementation and extensive experimentation.

Acknowledgements

This research has been supported by Yuan Shi of Temple University, who provided guidance, as well as access to the Synergy parallel application development environment.

The author is also grateful to David Wonnacott of Haverford College for his help in preparation of this manuscript, and to Nathan Doty for his review comments.

References

[1] G. Ciardo, A. Blakemore, P.F. Chimento, J.K. Muppala, and K.S. Trivedi. Automated generation and analysis of Markov reward models using stochastic reward nets. In C.D. Meyer and R.J. Plemmons, editors. Linear Algebra, Markov Chains, and Queueing Models. *IMA Volumes in Mathematics and its Applications*, 48: 145–191, 1993.

[2] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian, and T. von Eicken. LopP: A practical model of parallel computation. *Communications of the ACM*, 39, 11: 78–85, November 1996.

[3] N. Doty and J.P. Dougherty. Parallel application performance on a network of workstations. *Supercomputing'98*, Orlando, Florida, USA, November 1998.

[4] J.P. Dougherty. Monte Carlo integration in a distributed heterogeneous environment. In *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences*, Maui, Hawaii, USA, January 1993.

[5] J.P. Dougherty. Structured performability analysis of fault tolerant parallel and distributed applications. Ph.D. dissertation, Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania, USA, 170 pages, January 1998.

[6] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21, 9: 948–960, September 1972.

[7] P. Jalote. *Fault Tolerance in Distributed Systems.* Englewood Cliffs, New Jersey: Prentice Hall, 1994.

[8] P.G. Neumann. Distributed systems have distributed risks. *Communications of the ACM*, 39, 11: p. 130, November 1996.

[9] G.F. Pfister. *In Search of Clusters.* Englewood Cliffs, New Jersey: Prentice Hall, 1995.

[10] *Proceedings of the Second Annual IEEE International Computer Performance and Dependability Symposium.* Urbana-Champaign, Illinois, USA, September 1996.

[11] C.V. Ramamoorthy and W. Tsai. Advances in software engineering. *IEEE Computer*, 29, 10: 47–58, October 1996.

[12] R.A. Sahner and K.S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Transactions on Software Engineering*, 14, 10: 1105–1114, October 1987.

[13] S. Sahni and V. Thanvantri. Performance metrics: Keeping the focus on runtime. *IEEE Parallel and Distributed Technology*, pages 43–56, Spring 1996.

[14] Y. Shi. Parallel program scalability analysis. In *Proceeding of the Ninth IASTED International Conference on Parallel and Distributed Computing Systems*, Washington, D.C., USA, October 1997.

[15] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30, 2: 123–169, June 1998.

[16] K.S. Trivedi, G. Ciardo, M. Malhorta and R.A.Sahner. Dependability and performability analysis. In L. Donatiello and R. Nelson, editors. Performance evaluation of computer and communication systems. *Lecture Notes in Computer Science*, 729: 587–612, Springer-Verlag, 1993.

[17] P.R. Woodward. Perspectives on supercomputing: Three decades of change. *IEEE Computer*, 29, 10: 99–111, October 1996.

Sorting on Clusters of SMPs

David R. Helman and Joseph JáJá
Institute for Advanced Computer Studies & Department of Electrical Engineering,
University of Maryland, College Park, MD 20742 USA
Phone: + 301 405 6758, Fax + 301 314 9658
E-mail: {helman, joseph}@umiacs.umd.edu

Keywords: Parallel Algorithms, Generalized Sorting, Sorting by Regular Sampling, Parallel Performance

Edited by: Rajkumar Buyya and Marcin Paprzycki
Received: September 8, 1998 **Revised:** January 25, 1999 **Accepted:** February 5, 1999

We introduce an efficient algorithm for sorting on clusters of symmetric multiprocessors (SMPs). This algorithm relies on a novel scheme for stably sorting on a single SMP coupled with balanced regular communication on the cluster. The algorithm was implemented in C using POSIX threads and the SIMPLE library of communication primitives and run on a cluster of DEC AlphaServer 2100A systems. Our experimental results verify the scalability and efficiency of our proposed solution and illustrate the importance of considering both memory hierarchy and the overhead of shifting to multiple nodes.

1 Introduction

Clusters of symmetric multiprocessors (SMPs) have emerged as the primary candidates for large scale multiprocessor systems. In spite of this trend, relatively little work has been done to develop techniques for designing algorithms which make effective use of the resources available on such platforms. This task is made difficult by the contrasting requirements of the platform. On the one hand, each SMP must be viewed on its own as a hierarchical shared memory machine. Good performance requires both good load distribution and the minimization of main memory access. On the other hand, from the perspective of the cluster, each node is in effect a superproccessor, and therefore the cluster of SMPs is a collection of powerful processors connected by a communication network. Maximizing the performance of such a distributed memory machine requires both efficient load balancing and regular balanced communication.

In this paper, we examine the problem of sorting on SMP clusters. Sorting is arguably the most studied problem in computer science, due in large part to its pervasiveness. But it is also intrinsically interesting because of its demanding requirements for irregular memory access and interprocessor communication. From the perspective of the cluster, any algorithm which performs well for distributed memory machines would be a reasonable candidate. In particular, we have identified two such algorithms in our previous work [8, 11]. The first is a variation on sample sort and the other is a variation on the approach of sorting by regular sampling. On the other hand, from the perspective of the individual SMP, we might consider any algorithm which is designed for hierarchical shared memory machines. Unfortunately, none of these algorithms by itself is sufficient to achieve efficient performance on an SMP cluster. The reason for this is that algorithms for shared memory machines typically capitalize on the fact that accessing data associated with another processor is no more expensive than accessing its own data in the shared memory. Assuming the entire cluster is a shared memory platform will underestimate the cost of sharing data between nodes in the cluster. On the other hand, efficient algorithms for distributed memory machines tend to confine interprocessor communication to a minimum number of regular balanced exchanges. As such, assuming the entire cluster is a distributed memory platform will exaggerate the cost of sharing data on an SMP.

We introduce a sorting algorithm for clusters of SMPs which is a hybrid of our modified algorithms for parallel sorting by random sampling and parallel sorting by deterministic sampling. To our knowledge, this is the first sorting algorithm specifically designed for this platform. Our algorithm was coded in C using POSIX Threads and the SIMPLE library of communication primitives [3]. We examined its performance on a cluster of DEC AlphaServer 2100A systems linked by an ATM network, using a variety of benchmarks that we have identified to assess the dependence of our algorithm on the input distribution. Our experimental results verify the scalability and efficiency of our proposed solution and illustrate the importance of considering both memory hierarchy and the overhead of shifting to multiple nodes.

The organization of this paper is as follows. **Section 2** presents our computational model for analyzing

algorithms for SMP clusters. **Section 3** describes our sorting algorithm for this platform. Finally, **Section 4** describes the experimental performance of our sorting algorithm on an SMP cluster.

2 Computational Model

We measure the overall complexity of an algorithm by the triplet $\langle C_A, C_V, T_N \rangle$, where C_A is the maximum number of communication primitives called by any node, C_V is the maximum amount of data sent or received by any node, and T_N is the maximum time required by a node for local computation. Additionally, we insist that all internode communication must be balanced. The basis for this requirement is the experimental observation by both ourselves [4] and other workers (e.g. [6]) that large variations result in an inefficient use of the communication bandwidth. Further, utilizing regular communication has become more important with the advent of message passing standards, such as MPI, which seek to guarantee the availability of very efficient (often machine specific) implementations of certain basic collective communication routines. Note also that we report C_A and C_V as *approximations* of the actual values. By approximations, we mean that if C_A or C_V is described by the expression $\left(c_k x^k + c_{(k-1)} x^{(k-1)} + ... + c_0 x^0\right)$, then we report it using the approximation $\left(c_k x^k + o\left(x^k\right)\right)$. We report the communication cost in this fashion because of the dominant expense of internode communication in a distributed memory architecture. On the other hand, despite the importance of these memory costs, we report only the highest order term, since otherwise the expression can easily become unwieldy.

We measure T_N, the time required for local computation at a node, as follows. We view each node as a symmetric multiprocessor (SMP) consisting of a single level of cache and a shared main memory. In our model, we acknowledge the dominant expense of memory access. Indeed, it has been widely observed that the rapid progress in microprocessor speed has left main memory access as the primary limitation to SMP performance. The problem can be minimized by insisting where possible on a pattern of contiguous data access. This exploits the contents of each cache line and takes full advantage of the pre-fetching of subsequent cache lines. However, since it does not always seem possible to direct the pattern of memory access, our complexity model needs to include an explicit measure of the number of non-contiguous main memory accesses required by an algorithm.

More precisely, we measure the overall complexity at a node T_N by the triplet $\langle M_A, M_E, T_C \rangle$, where M_A is the maximum number of accesses made by any processor to main memory, M_E is the maximum amount of data exchanged by any processor with main memory, and T_C is an upper bound on the local computational complexity of any of the processors. Note that M_A is simply a measure of the number of non-contiguous main memory accesses, where each such access may involve an arbitrary sized contiguous block of data. While we report T_C using the customary asymptotic notation, we report M_A and M_E as *approximations* (see above) of the actual values. We report the memory access in this fashion because of the dominant expense of memory access on this architecture. With so few processors available, this coefficient is usually crucial in determining whether or not a parallel algorithm can be a viable replacement to the sequential alternative.

Hence, the overall complexity of an algorithm on a cluster of high performance nodes is given by the five values $\langle C_A; C_V; M_A; M_E; T_C \rangle$ (though, in practice, it is often possible to focus on a subset of these values). Note that our approach to designing algorithms for clusters of SMPs is distinct from models and methods that are currently being promoted. Both the bulk-synchronous parallel model [16] and the more recent Queue-Read Queue-Write (QRQW) PRAM model [7] have been promoted by the their authors as possible "bridging models" which can span the entire array of available architectures. The BSP model suggests that all platforms can be generalized as a message-passing distributed memory architecture. This idea has found expression in programming methodologies (e.g. [13]) which enforce a shared-nothing paradigm between tasks, and all communication and coordination between tasks are performed through the exchange of explicit messages, even tasks on a node with physically shared memory. On the other hand, the QRQW model suggests that all platforms can be generalized as a shared memory architecture. This idea has found expression in programming methodologies (e.g. [2]) which use a software layer to simulate coherent shared memory between nodes by transparently using messages to move around specific data or referenced memory pages. Both of these methodologies accept inefficiencies in order to simplify programmability and portability. The reason for this is that on an SMP accessing the data associated with another processor is no more expensive than accessing one's own data in main memory. On the other hand, accessing the data at another SMP is far more expensive then accessing one own data since it requires the use of explicit message passing. Thus, assuming that the entire platform is a distributed memory machine exaggerates the cost of sharing data between processors on an SMP, whereas assuming that the entire platform is a purely shared memory machine underestimates the cost in sharing data between nodes in the cluster. Hence, these current approaches lead to significant inefficiencies that will make them unacceptable for a wide range of problems.

3 Our Sorting Algorithms

Consider the problem of sorting n elements equally distributed amongst N nodes, where each node has p processors. Any algorithm which performs well on a distributed memory platform might seem a reasonable candidate for a cluster of SMPs. Prominent among them are sample sort [6] and parallel sorting by regular sampling [15], each of which requires only a single round of all-to-all communication. However, the price paid for a single step of communication is an irregular communication scheme and difficulty with load balancing. No matter how the routing is scheduled, there exist inputs that give rise to large variations in the number of elements destined for different nodes. In response, we have introduced two novel algorithms which address the limitations of these two algorithms. Our first algorithm [8] is a novel variation on (randomized) sample sort, and our second algorithm [11] is a novel variation on the approach of sorting by regular sampling. Both algorithms replace the single step of irregular communication with only two rounds of regular, balanced communication. Not only does this afford us efficient and predictable communication, but we leverage this modification to allow us to obtain two other important results. First, we are able to sustain a very high sampling ratio at virtually no cost, allowing us to minimize the problem of poor load balancing. And, second, we efficiently accommodate the presence of duplicate values without the overhead of tagging each element. The resulting algorithms appear to outperform all similar sorting algorithms on distributed memory platforms.

However, to be useful on a cluster of symmetric multiprocessors, we need to show that each step of our distributed memory algorithms can be replaced where appropriate by a multithreaded SMP implementation. In particular, we need to replace the sequential sort with an efficient shared memory algorithm. With this in mind, we will first describe a novel algorithm for sorted on a single SMP, and then we discuss how it is incorporated into an distributed memory algorithm to produce an efficient algorithm for sorting on clusters of SMPs.

3.1 Sorting on a Single SMP

Any of the algorithms that have been proposed in the literature for sorting on hierarchical memory models can be considered for possible implementation on an SMP. These include balance sort [14], sharesort [1], and simple randomized merge sort [5]. However, without modifications, most are unnecessarily complex or inefficient for a relatively simple platform such as ours. A notable exception is the algorithm of Varman et al. [17]. Yet another approach is an adaptation of our sorting by regular sampling algorithm [11], which we originally developed for distributed memory machines. The idea behind sorting by regular sampling is to first partition the n input elements into p memory-contiguous blocks and then sort each of these blocks using an appropriate sequential algorithm. Then, a set of $p-1$ *splitters* is found to partition each of these p sorted sequences into p subsequences indexed from 0 up to $(p-1)$, such that every element in the i^{th} group is less than or equal to each of the elements in the $(i+1)^{th}$ group, for $(0 \leq i \leq p-2)$. Then, the task of merging the p subsequences with a particular index can be turned over to the correspondingly indexed processor, after which the n elements will be arranged in sorted order. One way to choose the splitters is by regularly sampling the input elements - hence the name Sorting by Regular Sampling. As modified for an SMP, this algorithm is similar to the parallel sorting by regular sampling (PSRS) algorithm of Shi and Schaeffer [15]. However, unlike their algorithm, our algorithm accommodates the presence of duplicate values without the overhead of tagging each element.

While our algorithm will efficiently partition the work amongst the available processors, it will not be sufficient to minimize main memory accesses unless we also carefully specify how the sequential tasks are to be performed. Specifically, straightforward binary merge sort or quick sort will require $\log \frac{n}{p}$ memory accesses for each element to be sorted. Thus, a more efficient sequential sorting algorithm running on a single processor can be expected to outperform a parallel algorithm running on the relatively few processors available with an SMP, unless the sequential steps of the parallel algorithm are properly optimized. Knuth [12] describes a better approach for the analogous situation of external sorting. First, each processor partitions its $\frac{n}{p}$ elements to be sorted into blocks of size $\frac{C}{2}$, where C is the size of the cache, and then sorts each of these blocks using merge sort. This alone eliminates $\log\left(\frac{C}{4}\right)$ memory accesses for each element. Next, the sorted blocks are merged z at a time using a tournament of losers, which further reduces the memory accesses by a factor of $\log z$. To be efficient, the parameter z must be set less than $\frac{C}{L}$, where L is the cache line size, so that the cache can hold the entire tournament tree plus a cache line from each of the z blocks being merged. Otherwise, as our experimental evidence demonstrates, the memory performance will rapidly deteriorate.

The pseudocode for our algorithm is as follows:

- **(1)** Each processor P_i $(0 \leq i \leq p-1)$ sorts the subsequence of the n input elements with indices $\left(\frac{in}{p}\right)$ through $\left(\frac{(i+1)n}{p} - 1\right)$ as follows:
  - **(A)** Sort each block of m input elements using an appropriate sequential algorithm, where $m \leq \frac{C}{2}$. For integers we use the radix sort algorithm, whereas for floating point

numbers we use the merge sort algorithm.

- **(B)** For $j = 0$ up to $\left(\frac{\log(n/pm)}{\log(z)} - 1\right)$, merge the sorted blocks of size (mz^j) using z-way merge, where $z < \frac{C}{L}$.

- **(2)** Each processor P_i selects each $\left(\frac{in}{p} + (j+1)\frac{n}{ps}\right)^{th}$ element as a sample, for $(0 \leq j \leq s-1)$ and a given value of s $\left(p \leq s \leq \frac{n}{p^2}\right)$.

- **(3)** Processor $P_{(p-1)}$ merges the p sorted subsequences of samples and then selects each $((k+1)s)^{th}$ sample as Splitter[k], for $(0 \leq k \leq p-2)$. By default, the p^{th} splitter is the largest value allowed by the data type used. Additionally, binary search is used to compute for the set of samples with indices 0 through $((k+1)s-1)$ the number of samples Est[k] which share the same value as Splitter[k].

- **Step (4):** Each processor P_k uses binary search to define an index $b_{(i,k)}$ for each of the p sorted input sequences created in Step (1). If we define $T_{(i,k)}$ as a subsequence containing the first $b_{(i,k)}$ elements in the i^{th} sorted input sequence, then the set of p subsequences $\{T_{(0,k)}, T_{(1,k)}, ..., T_{((p-1),k)}\}$ will contain all those values in the input set which are strictly less than Splitter[k] and *at most* $\left(\text{Est}[k] \times \frac{n}{ps}\right)$ elements with the same value as Splitter[k]. The term *at most* is used because there may not actually be this number of elements with the same value as Splitter[k].

- **Step (5):** Each processor P_k merges those subsequences of the sorted input sequences which lie between indices $b_{(i,(k-1))}$ and $b_{(i,k)}$ using p-way merge. It is shown in [11] that no processor will merge more than $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ elements.

It is straightforward to see how this algorithm can be implemented as a stable integer sort, and [10] contains an informal proof. The analysis of our algorithm is as follows. In Step (1A), each processor moves through a contiguous portion of the input array to sort it in blocks of size m using an appropriate sequential sort algorithm. If we assume that $\left(m \leq \frac{C}{2}\right)$, this will require only a single non-contiguous memory accesses to exchange $\frac{2n}{p}$ elements with main memory and either $O\left(\frac{n}{p}\right)$ computation time to sort integers using radix sort or $O\left(\frac{n}{p}\log m\right)$ computation time to sort floating point values using merge sort. Step (1B) involves $\frac{\log(n/pm)}{\log(z)}$ rounds of z-way merge. Since round j will begin with $\frac{n}{pmz^j}$ blocks of size mz^j, this will require at most $\frac{2nz}{pm(z-1)}$ non-contiguous memory accesses to exchange $\frac{2n\log(n/pm)}{p\log(z)}$ elements with main memory memory and $O\left(\frac{n}{p}log\left(\frac{n}{pm}\right)\right)$ computation time. The selection of s noncontiguous samples by each processor in Step (2) requires s non-contiguous memory accesses to exchange $2s$ elements with main memory and $O(s)$ computation time. Step (3) involves a p-way merge of blocks of size s followed by p binary searches on segments of size s. Hence, it requires approximately $p\log(s)$ non-contiguous memory accesses to exchange approximately $2sp$ elements with main memory and $O(sp\log p)$ computation time. Step (4) involves p binary searches by each processor on segments of size $\frac{n}{p}$ and hence requires approximately $p\log\left(\frac{n}{p}\right)$ non-contiguous memory accesses to exchange approximately $p\log\left(\frac{n}{p}\right)$ elements with main memory and $O\left(p\log\left(\frac{n}{p}\right)\right)$ computation time. Step (5) involves a p-way merge of p sorted sequences whose combined length is at most $\left(\frac{n}{p} + \frac{n}{s} - p\right)$. This requires approximately p non-contiguous memory accesses to exchange approximately $2\left(\frac{n}{p} + \frac{n}{s}\right)$ elements with main memory and $O\left(\frac{n}{p}\log p\right)$ computation time. Hence, the overall complexity of our shared memory sorting algorithm is given by

$$\begin{aligned} T(n,p) &= \langle M_A; M_E; T_C \rangle \\ &= \left\langle \left(\frac{nz}{pm(z-1)} + s + p\log\left(\frac{n}{p}\right)\right); \right. \\ &\qquad \left(\left(2\frac{\log(n/pm)}{\log(z)} + 2\right)\frac{n}{p} + 2\frac{n}{s}\right); \\ &\qquad \left. O\left(\frac{n}{p}\log n\right)\right\rangle \end{aligned}$$

for $\left(p \leq s \leq \frac{n}{p^2}\right)$, $n \geq ps$, $m \leq \frac{C}{2}$, and $z \leq \frac{C}{L}$. Since the analysis suggests that the parameters m and z should be as large as possible subject to the stated constraints while selecting s so that $\left(p << s << \frac{n}{p^2}\right)$, we would expect that in practice the complexity of our algorithm could be characterized as

$$\begin{aligned} T(n,p) &= \langle M_E; T_C \rangle \\ &= \left\langle \left(2 + 2\frac{\log(n/pm)}{\log(z)}\right)\frac{n}{p}; O\left(\frac{n}{p}\log n\right)\right\rangle. \end{aligned}$$

3.2 Sorting On a Cluster of SMPs

Both our randomized sample sort algorithm [8] and our sorting by regular sampling algorithm [11] would be an appropriate choice for a cluster of SMPs, but we chose the randomized sample sort because it proved to be slightly faster in its implementation. We repeat the pseudocode here for convenience, where we replace each sequential step where appropriate by a multithreaded SMP implementation (note that the communication primitives mentioned are described in detail in [9]):

- **Step (1):** Using p threads, each node N_i $(0 \leq i \leq (N-1))$ randomly assigns each of its $\frac{n}{N}$ elements to one of N buckets. With high probability, no bucket will receive more than $c_1 \frac{n}{N^2}$ elements, where c_1 is a constant to be defined later.

- **Step (2):** Each node N_i routes the contents of bucket j to node N_j, for $(0 \leq j \leq (N-1))$. Since with high probability no bucket will receive more than $c_1 \frac{n}{N^2}$ elements, this is equivalent to performing a `transpose` operation with block size $c_1 \frac{n}{N^2}$.

- **Step (3):** Using p threads, each node N_i sorts the at most $\left(\alpha_1 \frac{n}{N} \leq c_1 \frac{n}{N}\right)$ values received in Step (2) with the appropriate version of our SMP sorting algorithm, depending on the data type.

- **Step (4):** From its sorted list of $\left(\gamma \frac{n}{N} \leq c_1 \frac{n}{N}\right)$ elements, node N_0 selects each $\left((j+1)\gamma \frac{n}{N^2}\right)^{th}$ element as Splitter$[j]$, for $(0 \leq j \leq N-2)$. By default, Splitter$[N-1]$ is the largest value allowed by the data type used. Additionally, for each Splitter$[j]$, binary search is used to determine the values $\text{Frac}_\text{L}[j]$ and $\text{Frac}_\text{R}[j]$, which are respectively the fractions of the total number of elements at node N_0 with the same value as Splitter$[j-1]$ and Splitter$[j]$ which also lie between index $\left((j-1)\gamma \frac{n}{N^2}+1\right)$ and index $\left(j\gamma \frac{n}{N^2}\right)$, inclusively.

- **Step (5):** Node N_0 `broadcasts` the Splitter, Frac_L, and Frac_R arrays to the other $(N-1)$ nodes.

- **Step (6):** Each node N_i uses binary search on its sorted local array to define for each of the N splitters a subsequence S_j. The subsequence associated with Splitter$[j]$ contains all those values which are greater than Splitter$[j-1]$ and less than Splitter$[j]$, as well as $\text{Frac}_\text{L}[j]$ and $\text{Frac}_\text{R}[j]$ of the total number of elements in the local array with the same value as Splitter$[j-1]$ and Splitter$[j]$, respectively.

- **Step (7):** Each node N_i routes the subsequence associated with Splitter$[j]$ to node N_j, for $(0 \leq j \leq (N-1))$. Since with high probability no sequence will contain more than $c_2 \frac{n}{N^2}$ elements, where c_2 is a constant to be defined later, this is equivalent to performing a `transpose` operation with block size $c_2 \frac{n}{N^2}$.

- **Step (8):** Using p threads, each node N_i merges the N sorted subsequences received in Step (7) to produce the i^{th} column of the sorted array. Note that, with high probability, no node has received more than $\alpha_2 \frac{n}{N}$ elements, where α_2 is a constant to be defined later.

Noting that we have established in [8] that with high probability $c_1 \geq 2$, $\alpha_2 \geq 2.62$, and $c_2 \geq 5.42$, the analysis of our sample sort algorithm is as follows. Step (1) is easily done with multiple threads by having each of the p processors at a node move through a contiguous block of the input array and randomly write each element to one of N blocks. This will require $(N+1)$ non-contiguous memory accesses to exchange $\frac{2n}{Np}$ elements with main memory and $O\left(\frac{n}{Np}\right)$ computation time. In Step (2), with high probability no two nodes will exchange more than $2\frac{n}{N^2}$ elements, and so it will be sufficient to use a `transpose` primitive which can transfer at most $\left(\frac{2n(N-1)}{N^2}\right)$ elements. However, it is important to note that a $\frac{N^2}{n}$-biased binomial process encounters *on average* $\frac{n}{N^2}$ successes in $\frac{n}{N}$ trials and so *in practice* it will be sufficient to use a `transpose` primitive which can transfer approximately $\left(\frac{n(N-1)}{N^2}\right)$ elements (which is what we observe experimentally in the next section). The cost of sorting at most $2\frac{n}{N}$ elements in Step (3) using multiple threads depends on the data type. Sorting integers using radix sort requires exchanging $\left(4\frac{\log(n/Npm)}{\log(z)}+4\right)\frac{n}{Np}$ elements with main memory and $O\left(\frac{n}{Np}\log p\right)$ computation time, whereas sorting doubles using merge sort requires exchanging $\left(4\frac{\log(n/Npm)}{\log(z)}+4\right)\frac{n}{Np}$ elements with main memory and $O\left(\frac{n}{Np}\log n\right)$ computation time. (assuming $\frac{n}{N} \geq ps$ and $\left(p << s << \frac{n}{Np^2}\right)$. Again, it is important to note that on *on average* each node only needs to sort approximately $\frac{n}{N}$ elements, which in turn requires exchanging only about $\left(2\frac{\log(n/Npm)}{\log(z)}+2\right)\frac{n}{Np}$ elements with main memory. Steps (4) and (6) each involve a single thread performing $2N$ binary searches on sequences of size $O\left(\frac{n}{N}\right)$. This requires approximately $2N\log\left(\frac{n}{N}\right)$ non-contiguous memory accesses to exchange approximately $2N\log\left(\frac{n}{N}\right)$ elements with main memory and $O\left(N\log\left(\frac{n}{N}\right)\right)$ computation time. Step (5) involves broadcasting $3N$ elements from node N_0 to the other $(N-1)$ processors. As discussed in [9], this can be efficiently implemented by first performing a `scatter` operation on these $3N$ elements followed by a `gather` operation at each of the nodes, which together requires transferring at total of at most $(6N)$ elements using balanced communication exchanges. In Step (7), with high probability no two nodes will exchange more than $5.42\frac{n}{N^2}$ elements, and so it will be sufficient to use a `transpose` primitive which can transfer at most $\left(\frac{5.42n(N-1)}{N^2}\right)$ elements. However, it is important to note that *on average* no two nodes will exchange more than $\frac{n}{N^2}$ elements and so *in practice* it will be sufficient to use a `transpose` primitive which can transfer approximately $\left(\frac{n(N-1)}{N^2}\right)$ elements (which is what we

observe experimentally in the next section). Finally, Step (8) involves merging p sorted sequences whose combined length is at most $2.62\frac{n}{N}$. This can be easily done using multiple threads by partitioning each sorted sequence into p blocks using the same scheme used in our algorithm for sorting on a shared memory platform. This requires requires exchanging $\frac{5.24n}{Np}$ elements with main memory and $O\left(\frac{n}{Np}\log p\right)$ computation time (assuming $\left(p << s << \frac{n}{Np^2}\right)$ and $\frac{n}{N} \geq ps$). Again, it is important to note that on *on average* each node only needs to merge approximately $\frac{n}{N}$ elements, which in turn requires exchanging only about $2\frac{n}{Np}$ elements with main memory. Hence, with high probability, the overall complexity of our sorting algorithm is given (for floating point numbers) by

$$\begin{aligned} T(n,N,p) &= \langle C_A; C_V; M_A; M_E; T_C \rangle \\ &= \left\langle 4; 7.24\frac{n}{Np}; 2N\log\left(\frac{n}{N}\right); \right. \\ &\qquad \left(9.24 + 4\frac{\log(n/Npm)}{\log(z)}\right)\frac{n}{Np}; \\ &\qquad \left. O\left(\frac{n}{Np}\log n\right)\right\rangle \end{aligned}$$

for $N^2 < \frac{n}{3\ln n}$, $\left(p << s \leq<< \frac{n}{Np^2}\right)$, and $\frac{n}{N} \geq ps$. Noting that the M_A non-contiguous memory accesses comprise a insignificant proportion of the M_E total elements exchanged with memory, and recalling that *on average* each processor traverses only about $\frac{n}{p}$ elements in Step (4), we would expect that *in practice*, the complexity of our sample sort algorithm will be approximately:

$$\begin{aligned} T(n,N,p) &= \langle C_A; C_V; M_E; T_C \rangle \\ &= \left\langle 4; 2\frac{n}{Np}; \left(4 + 2\frac{\log(n/Npm)}{\log(z)}\right)\frac{n}{Np}; \right. \\ &\qquad \left. O\left(\frac{n}{Np}\log n\right)\right\rangle. \end{aligned}$$

4 Performance Evaluation

Our algorithms were implemented using POSIX threads and run on a DEC Alpha Cluster. Our DEC Alpha cluster consists of 10 AlphaServer 2100A systems, each of which holds 4 Alpha 21064A processors running each at 275 MHz. Each Alpha 21064A processor has a 16KB primary data cache and a 4MB secondary data cache. The AlphaServers are connected using the Digital Gigaswitch/ATM and OC-3c adapter cards, which have a peak bandwidth rating of 155.52 Mbps. Internode communication is effected by calls to the SIMPLE collective communication primitives [3].

We tested our code on a variety of benchmarks, each of which had both a 32-bit integer version and a 64-bit double precision floating point number (double) version. See [9] for a detailed description and justification of these benchmarks.

4.1 Experimental Results for a Single SMP

For each experiment, the input is a single array of elements, and the output is these elements arranged in a single array in non-descending order. Table 4.1 verifies that as expected performance does not significantly depend on the input distribution. Because of this independence, the remainder of this section will only discuss performance on the single benchmark [U], in which the input data forms a uniform random distribution.

Table 2 displays the times required to sort 4M *doubles* (i.e. double precision floating point values) using a single thread as a function of m and z. Notice first that performance suffers dramatically when the block size reaches 4MB (512K eight byte double precision numbers), which is the limit of the cache on the AlphaServer. But consider the data for a given block size - say 2K. The execution time drops as we move from $z = 2$ to $z = 64$. This is reasonable since we require 11 rounds of 2-way merge, 6 rounds of 4-way merge, 4 rounds of 8-way merge, 3 rounds of 16-way and 32-way merge, and only 2 rounds of 64-way merge, and each round of z-way merge is obviously another round where all the input elements must be brought in from main memory. We would then expect that moving from $z = 64$ to $z = 1024$ would have little effect on the execution time since it does nothing to reduce the memory requirements, but this turns out not to be the case. The explanation lies in recalling that the AlphaServer has both a primary and a secondary cache. An efficient implementation of the z-way merge in Step (1B) would fill this 16 KB primary cache with the entire tree of losers (z 12 byte records) plus a cache line (32 bytes) from each of the z sequences being merged. For $z = 256$, this primary cache is essentially filled, and cache misses to secondary cache become an issue. Finally, note the difference between the optimal sorting time of 10.28 seconds for $m = 16K$ and $z = 256$ with the time of 18.96 seconds required to sort using only binary merge sort. Here, reducing memory access by a combination of block sorting and z-way merging improved the performance by 45%. Such results strongly support the attention that we place in this algorithm on the volume of main memory accesses.

Figure 1 examines the scalability of our sorting algorithm as a function of the number of threads, for different problem sizes. Bearing in mind that these graphs are log-log plots, they show that for a *fixed input size* n the execution time nearly halves when the number of threads p is doubled.

Input Size	Benchmark					
	[U]	[G]	[Z]	[WR]	[DD]	[RD]
512K	0.397	0.394	0.320	0.421	0.337	0.348
1M	0.868	0.856	0.741	0.844	0.724	0.710
2M	1.64	1.72	1.39	1.73	1.40	1.51
4M	3.50	3.47	3.00	3.52	3.01	2.98

Table 1: Sorting doubles (in seconds) using 4 threads.

Block Size	Denomination of z-Way Merge						
	2	8	32	64	256	1024	2048
2K	18.45	12.09	11.53	11.21	12.18	13.09	11.51
4K	17.29	12.11	10.67	11.21	12.31	11.24	
8K	16.47	11.33	10.66	11.31	12.19		
16K	15.61	11.37	10.79	11.44	10.28		
32K	15.04	11.71	11.11	11.52			
64K	14.54	10.99	11.49	10.59			
128K	14.91	12.19	11.33				
256K	15.38	13.63					
512K	18.36	16.52					
1M	19.17						
2M	18.87						
4M	18.96 - (No z-way merge is necessary for this block size)						

Table 2: Time (in seconds) required to sort 4M doubles using a single thread as a function of M and z.

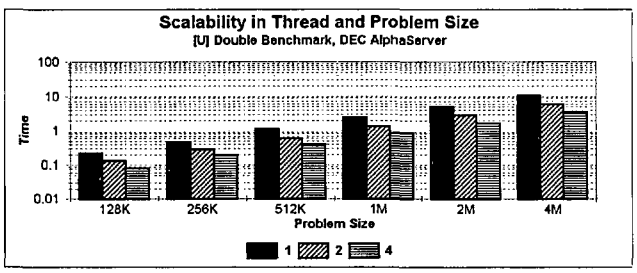


Figure 1: Scalability of sorting doubles with respect to the number of threads, for differing problem sizes.

4.2 Experimental Results for a Cluster of SMPs

For each experiment, the input is evenly distributed amongst the nodes. The output consists of the elements in non-descending order arranged amongst the nodes so that the elements at each node are in sorted order and no element at node N_i is greater than any element at processor N_j, for all $i < j$. Note that in all cases *the results shown for a single node were obtained using the sorting algorithm for a single SMP.*

Table 3 displays the performance of our sorting algorithm as a function of input distribution for a variety of input sizes. In each case, the performance is essentially independent of the input distribution. Because of this independence, the remainder of this section will only discuss the performance of our sorting algorithm on the single benchmark [U] (uniform distribution).

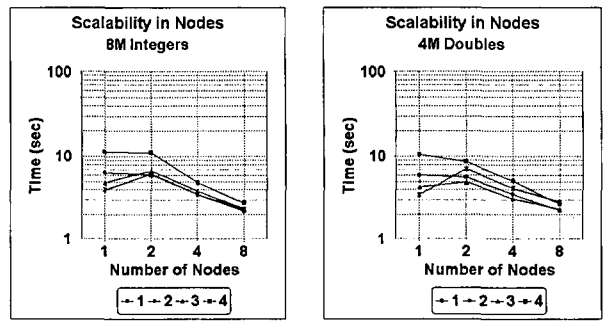


Figure 2: Scalability of sorting integers and doubles with respect to the number of nodes, for differing numbers of threads.

The results in **Figure 2** examines the scalability of our sorting algorithm as a function of the number of nodes, for a variety of threads. To understand these results, consider the step by step breakdown of the execution times shown in **Table 4** for sorting 8M integers with both 1 and 4 threads. Moving from one node to two introduces the overhead of Steps 1-2 and 4-8, which together account for approximately 35% and 50% of the total execution time on one node with 1 and 4 threads, respectively. This consumes the majority of the time we could hope to save by sharing the work of sorting amongst two nodes. The effect is more pronounced for multiple threads because as our model predicts internode communication is independent of

Input Size	Benchmark						
	[U]	[G]	[2-G]	[B]	[S]	[DD]	[RD]
4M	2.76	2.79	2.72	2.74	2.73	2.64	2.60
8M	4.89	4.86	4.80	4.76	4.85	4.61	4.54
16M	9.36	9.54	9.30	9.19	9.28	9.01	8.90
32M	18.71	19.31	18.68	18.27	18.54	18.23	18.31

Table 3: Total execution time (in seconds) required to sort a variety of double benchmarks on an 8 node cluster using 4 threads.

the number of threads. The effect of this overhead would be even more pronounced were it not for the fact that the time required for Step 3 for both 1 and 4 nodes is considerably higher than we would expect from sorting 4M integers on a single node. But moving between 1 and 4 nodes and 1 and 8 nodes, the time required for Step (3) scales inversely with the number of nodes, which is the expectation of our model. The failure of communication in Steps 2 and 7 to scale inversely with the number of nodes might at first appear surprising. However, this performance is actually quite reasonable if we recall that for 2, 4, and 8 nodes, each node has to send approximately 2M, 1.5M, and 0.875M integers across the network, respectively. The clear implication of these results is that an algorithm must be both efficient and scalable to justify the use of multiple nodes.

Step(s)	One Thread			
	1	2	4	8
1	0.00	0.87	0.41	0.22
2	0.00	1.28	0.92	0.56
3	11.19	7.17	2.39	1.17
4-6	0.00	0.00	0.00	0.00
7	0.00	1.24	0.84	0.49
8	0.00	0.49	0.27	0.32
Total	11.19	11.05	4.83	2.76

Step(s)	Four Threads			
	1	2	4	8
1	0.00	0.56	0.26	0.11
2	0.00	1.11	0.88	0.58
3	3.99	3.11	0.87	0.73
4-6	0.00	0.00	0.00	0.00
7	0.00	1.08	1.39	0.62
8	0.00	0.23	0.13	0.25
Total	3.99	5.09	3.53	2.29

Table 4: Time required for each step of sorting 8M integers with respect to the number of nodes using 1 and 4 threads.

The graph in **Figure 3** examine the scalability of our sorting algorithm as a function of problem size, for differing numbers of nodes and for 1 and 4 threads. For one thread, they show that for a *fixed number of nodes* there is an almost linear dependence between the execution time and the total number of elements n. The results for 4 threads are seemingly more problematic. However, a step by step breakdown of the execution times for sorting integers in [10] shows that the communication costs dominate the execution time and that as the problem size increases from 1M to 2M integers *the communication costs actually decrease*, presumably because of a change in the communication protocol. Once the protocol is switched, the relative costs of communication decline and the execution time scales with problem size as our model anticipates.

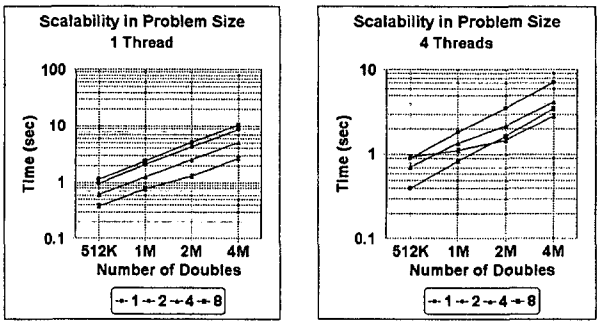


Figure 3: Scalability of sorting integers and doubles with respect to the problem size, for differing numbers of nodes and threads.

5 Conclusion

In this chapter, we introduce a efficient algorithm for generalized sorting on clusters of symmetric multiprocessors. To our knowledge, this algorithm is the first sorting algorithm specifically designed for this platform. Our algorithm was implemented and experimentally shown to be scalable in both the problem size and the number of nodes. As suggested by our computational model, our results illustrate the importance of considering both memory hierarchy and the overhead of shifting to multiple nodes when designing efficient algorithms for this platform.

References

[1] A. Aggarwal and G. Plaxton. Optimal Parallel Sorting in Multi-Level Storage. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 659–668, 1994.

[2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared Memmory Computing on Networks of Workstations. *IEEE Transactions on Computers*, 29(2):18–28, 1996.

[3] D.A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors. CS-TR-3798 and UMIACS-TR-97-48 Technical Report, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, May 1997.

[4] David A. Bader, David R. Helman, and J. JáJá. Practical Parallel Algorithms for Personalized Communication and Integer Sorting. *ACM Journal of Experimental Algorithmics*, 1(3):1–42, 1996.

[5] R. Barve, E. Grove, and J. Vitter. Simple Randomized Mergesort on Parallel Disks. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 109–118, Padua, Italy, June 1996.

[6] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha. A Expermental Analysis of Parallel Sorting Algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.

[7] P.B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. *SIAM Journal on Computing*, 1997. To appear.

[8] David R. Helman, David A. Bader, and J. JáJá. A Randomized Parallel Sorting Algorithm With an Experimental Study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998.

[9] D.R. Helman. *On the Design and Analysis of Practical Combinatorial Algorithms for Multiprocessor Architectures.* PhD thesis, Department of Electrical Engineering, University of Maryland, College Park, MD, 1998.

[10] D.R. Helman and J. JáJá. Sorting on Clusters of SMPs. Technical Report CS-TR-3833 and UMIACS-TR-97-69, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, July 1997.

[11] D.R. Helman, J. JáJá, and D.A. Bader. A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation. *ACM Journal of Experimental Algorithmics*, 3(4):1–24, 1998.

[12] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1973.

[13] S.S. Lumetta, A.M. Maimwaring, and D.E. Culer. Multi-Protocol Active Messages on a Cluster of SMPs. In *Proceedings of Supercomputing '97*, San Jose, CA, November 1997.

[14] M. Nodine and J. Vitter. Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, Velen, Germany, June 1993.

[15] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[16] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[17] P. Varman, B. Iyer, D. Haderle, and S. Dunn. Parallel Merging: Algorithm and Implementation Results. *Parallel Computing*, 15:165–177, 1990.

Problem Definition, Data Cleaning, and Evaluation: A Classifier Learning Case Study

Foster Provost
Bell Atlantic Science & Technology
500 Westchester Ave., White Plains, NY 10604, USA
Phone: 914-644-2169, Fax: 914-949-9589
E-mail: provost@acm.org
AND
Andrea Pohoreckyj Danyluk
Williams College, Department of Computer Science
Williamstown, MA 01267, USA
Phone: 413-597-2178, Fax: 413-597-4116
E-mail: andrea@cs.williams.edu

Keywords: classifier learning, evaluation, data engineering

Edited by: Xindong Wu
Received: August 23, 1996 **Revised:** September 9, 1998 **Accepted:** December 2, 1998

Problem definition, data cleaning, and evaluation constitute much of the process of building useful, real-world classifiers with inductive algorithms. This paper is a case study of this process based on a long-term project addressing the automatic dispatch of technicians to fix faults in the local loop of a telephone network. The bottom line of the project is that simple learning techniques can be effective. However, constructing a convincing argument to that effect is far from simple. In particular, we had to consult multiple sources to obtain class labels, use domain knowledge to clean up data, compare with existing methods, and evaluate with data from multiple locations. Finally, it was necessary to use decision-analytic techniques to evaluate the cost-effectiveness of the learned classifiers, because evaluation based on classification accuracy is misleading without an analysis of cost-effectiveness. Our view is that application studies should be helpful in guiding future research. Therefore, we conclude by outlining useful directions suggested by our experience on this long-term project.

1 Introduction

This paper presents a case study into the process of real-world classifier learning. The case study has been taken from the long-term MAX project, which addresses the automatic dispatch of technicians to fix faults in the local loop of a telephone network. For this paper, we use the term *machine learning* to denote the automatic generation of local-loop dispatch classifiers from historical data.

In the MAX domain we wish to learn classifiers for dispatching technicians to troubleshoot telephone line problems reported by phone company customers. In this domain, a small increase in accuracy can have a large impact on the company's bottom line. For example, if we are willing to ignore details for the moment, New York State alone has over three million residential trouble reports per year. If an erroneous dispatch costs the company (on average) $100, then even a one-percentage-point decrease in dispatch error rate can save the company over $3 million annually. Therefore, it is worthwhile to investigate methods for increasing the effectiveness of local-loop trouble diagnosis.

We discuss several interrelated issues involved in determining the efficacy of inductive learning programs in this domain. As the case study will show, such a determination is more complicated than merely running a handful of learning programs on a data set and comparing the accuracies of the resulting classifiers. To be convincing, we had to use multiple sources for class labels, and use domain knowledge to clean the data. For the first set of experiments presented, we use error rate as our primary metric. However, absolute error rates are not useful for comparisons between the experiments because the problem formulation varies. To facilitate inter-study comparison, we also report the percentage decrease in error rate (PDER) as compared to classifying all instances as the most frequently occurring class (the *default class*).

Next we discuss comparisons with existing methods, including both a set of experts and an existing expert system, as well as comparisons with data drawn from geographically disparate locations. The breadth

of this comparison study increased our confidence in our evaluation. Finally, we discuss that an error-rate comparison, albeit a fine starting point, is not sufficient for classifier evaluation in this real-world domain. What is important is the cost-effectiveness of the system, rather than its accuracy. Moreover, we show that a naive evaluation of cost-effectiveness also is not satisfactory, so we utilize techniques from decision analysis. From this case study a cost-sensitive learning method emerges as the most effective technique.

Our view is that application studies help to guide future research. Therefore we conclude by presenting a summary of general lessons learned and by outlining useful research directions suggested by our experience on this long-term project.

2 MAX and Machine Learning

MAX (Rabinowitz et al. 1991) is an expert system developed by NYNEX[1] Science and Technology for the purpose of troubleshooting customer-reported telephone problems. MAX deals specifically with problems in the local loop, the part of the telephone network between the central office and the customer's premises.

When a customer has difficulty with his telephone line he calls the phone company to report the problem (the *trouble*). A phone company representative creates a trouble report and also initiates electrical tests on the customer's line, called the Mechanized Loop Test (MLT).[2] The MLT measures the electrical signature of the customer's line and gives such information as voltages and resistances. All this information is then sent to a Maintenance Administrator (MA) who determines a high-level diagnosis for the trouble, and dispatches a technician to fix it. MAX (Maintenance Administrator eXpert) plays the role of an MA. It gives a high-level diagnosis of a trouble based upon MLT results, and other information about the customer. MAX can take one of five possible actions.

1. dispatch a cable technician (PDF);
2. dispatch an outside repair technician to the distribution wiring or customer premises (PDO);
3. dispatch a technician to the central office (PDI);
4. queue the trouble for further testing (PDT);
5. send the trouble to a human MA for diagnosis (PSH).

The problem of local-loop diagnosis was a particularly promising machine learning application for the following reasons.

1. Diagnosis in this domain is a static problem, i.e., all data are gathered and the dispatch decision is based on the values given. Difficult problems such as incorporating time are not an issue.
2. Data are abundant.
3. A knowledge base already exists, providing a wealth of information about the domain.
4. Small decreases in error rate can have a large impact.

Machine learning is also appealing because of its potential for generating dispatch knowledge that captures local differences and because of its potential for tracking changes in dispatch knowledge as the network equipment degrades or is replaced.

Several approaches to the problem of automatically generating dispatch knowledge from data have been investigated: (1) The application of inductive learning to generate completely new knowledge bases for specific locations (Danyluk & Provost 1993a,b). (2) The application of analytic and inductive learning to modify the existing knowledge base for specific locations (Pazzani & Brunk 1993). (3) The application of techniques to perform parameter tuning (Merz, et al., 1996). This paper discusses the first of these only.

Unless stated otherwise, all results reported in this paper were generated using the C4.5 decision tree learner (Quinlan 1993) with default settings.[3] Results given are after pruning. Numbers of test examples are given with each set of runs. All results reported have been averaged over 10 runs with independent training and test sets chosen randomly. Unless indicated otherwise, all data used in the runs in this paper are taken from a single site during a period of approximately eight months, and are described by 22 features used by MAX.

3 Multiple Data Sources

Determining whether learning programs can produce effective classifiers in this domain is complicated by a general belief that it is very difficult to ascertain the "correct" dispatches for historical trouble records, which has led to a general distrust of the class labels of the examples. To produce a robust evaluation we considered three different sources for the class labels, each of which created a slightly different learning problem.

First, we used MAX to generate class labels. If one assumes that MAX is performing the task satisfactorily, the ability to learn to duplicate MAX's performance is solid evidence that machine learning ap-

[1] Now Bell Atlantic. At the time MAX was developed, NYNEX was the parent company of New England Telephone and New York Telephone.

[2] MLT is a product of AT&T.

[3] Earlier results were obtained with other learning techniques, including rule learners and neural network learners, but C4.5 consistently has yielded results that are at least as good as the other systems.

proaches can be effective in generating dispatch classifiers from clean data. Second, we had experts generate class labels. If one assumes that the experts have knowledge not yet captured in MAX, then it would be useful to be able to model the classification performance of experts.

Finally, we generated class labels by cross-referencing a database of the resolutions reported by technicians in the field. The class labels generated from these resolutions are considerably noisy, due to errors in reporting and ambiguities in translation.[4] However, the ability to learn high-quality classifiers from these data would be very useful, because the potential exists to learn classifiers that capture knowledge unknown to the experts, and because the volume of data is potentially very large.

For the experiments reported in this section and the next, we evaluate learning results in two ways: (i) we measure error rate on independent test sets; and (ii) we measure the percentage decrease in error rate (PDER) of the learned concept description over the error rate of the default class. The PDER indicates the extent to which the learned decision tree decreases the error rate that would result from classifying all cases identically using the class that occurs most frequently in the training data (for which we use "the default class" as a shorthand).

The PDER is important because different data sets have different numbers of classes. MAX, for instance, has the option of diagnosing cases as being the type that need to be looked at by a human expert. These tend to be cases where the data required to analyze the trouble are missing. Field technicians, on the other hand, are not allowed such latitude. Therefore, a data set obtained from MAX will have more diagnostic class options than a data set of troubles analyzed by field technicians. Moreover, the machine learning studies also have different sets of dispatch options, which are described in detail below. Because different data sets have different numbers of classes, comparisons of absolute error rate are affected and the PDER becomes an important measure of the relative quality of the learned diagnostic knowledge.

3.1 Class labels obtained from MAX

As reported previously (Danyluk & Provost 1993a,b), we used the existing MAX expert system to create a "clean" data set from which to learn. We ran a series of experiments with the goal of showing that given good data we could learn knowledge to recreate MAX's behavior. We found that given a large enough quantity of data, using machine learning we can duplicate MAX's performance very well. As shown in Table 1, training on 1000 troubles yields an error rate of 0.09; training on 5000 troubles yields an error rate of 0.04. Although these results show promise for machine learning as a method of creating the knowledge base for a dispatch system, they do not offer a solution to the problem of generating knowledge that will increase the performance of MAX.

[4]Some of the codes used to describe resolutions do not map uniquely to dispatch classes.

Table 1: MAX data; five class problem. Size of test set = 871. Average error rate (ER) with the default class (PDT) = 0.54.

Training	ER	StDev	Avg PDER	StDev
100	.34	.04	36.51	7.92
500	.14	.02	73.44	3.90
1000	.09	.01	82.41	2.58
2000	.06	.01	89.31	2.61
5000	.04	.01	93.62	1.53

3.2 Class labels obtained from experts

In order to evaluate the potential of machine learning as a tool to build *a better* MAX, we enlisted the help of several experts in local-loop troubleshooting. The experts were phone company veterans with many years of experience in the areas of maintenance and repair of the local loop. We ran a set of experiments testing the ability to learn dispatch knowledge from expert-classified data. The rationale behind this set of experiments is that if machine learning can create knowledge that models the behavior of human experts well, then it may be possible, albeit resource consuming, to have local experts analyze large numbers of troubles and then to learn classifiers from these data.

Table 2 shows results for one expert who analyzed 500 troubles from one site. The results show that C4.5 can model the expert's behavior fairly well as compared to the default. Similar analyses of other experts' answers yielded comparable results. The large PDER suggests the potential for learning programs to model the behavior of human experts. Unfortunately, the size of the data set in these experiments was limited due to the limited availability of experts. The previous results of modeling MAX suggest that 400 examples may be too few for effective learning. An analysis of the classifiers learned from the MAX data explains why many examples are needed: very small disjuncts comprise a large portion of the concept description (Danyluk & Provost 1993a). Large data sets are necessary to learn small disjuncts with confidence (Provost & Aronis 1996).

Our intention had been to increase the volume of data by using multiple experts to generate a larger expert-classified data set. However, this exercise revealed that there is not a high degree of agreement

Table 2: Expert data; five class problem. Size of test set = 100. Average error rate (ER) with the default class (PDT) = 0.58.

Training	ER	StDev	Avg PDER	StDev
100	.39	.04	32.88	8.59
400	.35	.04	38.75	6.60

among experts as to the correct classification for a trouble. In fact, the error rate of the classifiers learned from the expert data was approximately equal to the error rate obtained when one expert was used to generate class labels for the evaluation of another expert. This suggests that the problem is much more difficult than previously thought. It also offers an explanation for the general distrust of class labels.

3.3 Class labels obtained from field technicians

The third data source from which we obtained class labels for troubles is the reporting of field technicians who fix ("resolve") the troubles. In order to generate class labels, we translated their resolution codes into the corresponding dispatches using a standard mapping. As the results in Table 3 show, the performance of the learned decision trees is less than inspiring. However, the learned trees do perform slightly better than the default.

Table 3: Technicians' data; four class problem. Size of test set = 863. Average error rate (ER) with the default class (PDF) = 0.62.

All Features				
Training	ER	StDev	Avg PDER	StDev
100	.61	.03	0.96	5.72
500	.60	.02	3.73	3.90
1000	.59	.01	4.51	2.53
5000	.58	.01	6.56	2.68

Vercode Only				
Training	ER	StDev	Avg PDER	StDev
100	.62	.04	-1.31	7.03
500	.54	.01	11.63	2.80
1000	.53	.02	13.24	2.80
5000	.52	.01	15.98	2.10

Quite surprisingly, we were able to increase significantly our ability to dispatch accurately by reducing the feature set to a single feature: *vercode*. Reducing the feature set to a single feature produces decision stumps, i.e., decision trees that split on a single feature only (Holte 1993). Vercode, generated by MLT, is a summary of the electrical readings into 50–150 categories; the decision stumps therefore have 50–150 leaves. As the results in Table 3 show, the decision stumps learned by C4.5 on the field data have higher accuracy than the decision trees learned with larger feature sets.

4 Cleaning up the Data

The experiments with class labels generated by MAX suggest considerable promise for machine learning in this domain. The experiments with class labels generated by the experts suggest that it is possible to model expert behavior to some degree, but that small expert-classified data sets are not sufficient to model expert behavior with high accuracy. Moreover, the disagreement among experts suggests that even if a given expert's behavior can be modeled with high accuracy, there will still be questions about the expert's performance. The most promising source of class labels is the field technician database. This database is very large and (arguably) based on fact rather than conjecture. Unfortunately, the learning programs had the most difficulty modeling these data. This almost certainly is because, with the given set of features, MAX generates class labels deterministically (and probably so do the experts), while the technicians' class labels are inherently probabilistic.

Analyzing the different trouble resolutions reported by the field technicians suggests some concrete reasons why machine learning programs would have a difficult time modeling the data. For some borderline resolutions at the interface between the cable and the distribution wiring, it is not clear what the correct dispatch should have been because the diagnosis cannot be mapped to a dispatch unambiguously. Furthermore, there are many cases for which the resolution is a "Test OK." This resolution indicates that the technician retested the line in the process of attempting to locate the trouble, and found that there was no longer a problem. Unfortunately, it is impossible to tell the difference between cases where there was no longer a problem to fix (e.g., the customer's second phone had been off the hook and was subsequently placed back on) and cases where the manifestation of the problem was transient (e.g., the trouble had been a short circuit due to the presence in a cable of water that had dried by the time the technician retested the line). Thus determining what the correct dispatch should have been is difficult.

We wanted to evaluate whether increasing the quality of the field data would improve the ability of a learning program to produce accurate classifiers. To this end, we used prior knowledge of trouble resolutions and dispatches to clean up the field data. Specif-

ically, we eliminated from the data all troubles for which the resolution was "Test OK." Additionally, we removed cases where it was impossible to determine from the resolution codes the correct dispatch, especially borderline cases. The effect of the data cleaning was to provide us with a set of cases for which we have only three class labels (PDF, PDO, PDI), but for which we have (relatively) high confidence in the correctness of those labels. We now describe a set of experiments that investigate the effect on learning of cleaning up the data.

Table 4: Cleaned data; three class problem. Size of test set = 686. Average error rate (ER) with the default class (PDF) = 0.47.

All Features				
Training	ER	StDev	Avg PDER	StDev
100	.41	.04	12.04	7.52
500	.38	.03	19.31	6.94
1000	.37	.02	20.72	3.94
2000	.36	.02	23.23	3.25

Vercode Only				
Training	ER	StDev	Avg PDER	StDev
100	.38	.04	18.97	10.07
500	.35	.02	26.48	3.90
1000	.35	.01	26.73	2.87
2000	.34	.02	27.44	4.15

As the learning results in Table 4 show, the performance on the cleaned-up data is considerably better than the performance on the original field data. It is important to note that the cleaned-up data have only three classes instead of four, and using the default yields a lower error rate than on the previous data. However, as the results in Table 4 show, the percentage decrease in error rate (PDER) for the learned concept descriptions is larger than with the original data.

These results provide support for the conclusion that from clean field data it is possible to learn more accurate classifiers. It must be noted, however, that by separating out the cases for which the final resolution is unambiguous, we may also be separating out the cases that are "easy" to diagnose. The effect of using this learned knowledge on the entire spectrum of troubles is still an open question, made very difficult by our inability to know the "correct" answer.

It should be noted that an alternative problem redefinition may also be effective. Specifically, much of the aforementioned ambiguity can be eliminated by combining two of the three dispatch classes. PDF (dispatch to a cable technician) and PDO (dispatch to an outside repair technician) both address problems in the "outside plant." There are *a priori* reasons why it might be desirable to combine these classes into a single "dispatch out" class. For example, training technicians to handle a larger class of problems may eliminate the need to separate problems in the outside plant. In order to test the hypothesis that we could differentiate accurately between dispatching "in" to the central office and dispatching "out" to the outside plant, we combined the two outside plant dispatches in the cleaned-up dataset. In doing so, we were able to reinsert those troubles eliminated because of PDF/PDO ambiguity. The results for the two-class problem are given in Table 5. As the table shows, the performance of the learned decision models is considerably better than the default when trained on large (2000 examples) data sets. In the table we report particularly high standard deviations for PDER in two cases. Inspection of the 10 runs shows that in two cases, the learned model performed similarly to the default, but in the remaining eight, it outperformed the default significantly.

Table 5: In vs out; two class problem. Size of test set = 738. Average error rate (ER) with the default class (PDO) = 0.09.

All Features				
Training	ER	StDev	Avg PDER	StDev
100	.09	.01	0.35	1.04
500	.09	.01	-0.50	2.11
1000	.08	.02	10.12	13.29
2000	.07	.01	24.89	2.93

Vercode Only				
Training	ER	StDev	Avg PDER	StDev
100	.09	.01	0.15	0.19
500	.09	.01	0.15	0.19
1000	.09	.01	0.15	0.19
2000	.07	.01	21.41	11.70

5 Comparison with Existing Methods

In the previous sections we compared the ability of learning programs to produce accurate classifiers from several different perspectives. The use of the field data as the source of class labels allows us to compare the performance of the learned classifier with the performance of MAX (and with the performance of the experts). Such a comparison has been a major component of Bell Atlantic's evaluation of the potential for learned knowledge to help with local-loop dispatch.

Table 6 compares the performance of the vercode

decision stumps with the MAX expert system on the three different versions of the field data (discussed above).[5] The comparison is complicated because MAX does not give solid dispatches on all the cases; it routes some difficult cases to a human analyst, and for others it requests additional tests. The decision stumps, on the other hand, produce a dispatch for every case.

It is important that we be as fair as possible in our comparison of the learned decision stumps and MAX. It is inappropriate to assume that MAX is in error each time it routes a trouble to an analyst. On the other hand, it is unfair to penalize the learned decision stump for being forced to make a decision on all cases. In order to make the comparison equitable, Table 6 reports

- error rates for MAX and for the learned decision stump (LDS) on all the test data
- error rate for MAX on the subset of cases for which it chose to make a dispatch (MAX-D)
- error rate for the learned decision stump on the subset of cases for which MAX made a dispatch (LDS-D)
- error rate for the learned decision stump on the subset of cases for which it was confident (LDS-C).[6]

Table 7 gives the sizes of the subsets, as a percentage of the entire dataset.

As Table 6 shows, with little exception, the learned decision stump outperforms MAX. The increase in performance using the learned vercode mapping over the MAX system is one piece of evidence supporting the conclusion that by looking at the data we can extract dispatch knowledge that can improve MAX's performance.

A potential criticism of the above argument is that the learning is fitting systematic error in the data (and that MAX actually may be as good or better at dispatching). Support for the contention that the learned knowledge is *not* just modeling errors in the data comes from a comparison of the effectiveness of the learned knowledge for dispatch in other geographic areas. In order for the effect of modeling systematic error to generalize across locations, the error must be systematic throughout the company. Furthermore, since we are using a vercode decision stump, the error must be systematic with respect to the vercode alone. We believe that this combination is highly unlikely.

[5] A comparison with the experts is not included in the summary, because the small number of troubles analyzed by the experts makes the performance of the experts incomparable.

[6] In all cases, a decision was deemed "confident" only if the estimated probability of membership in the predicted class was at least 0.6.

To test the hypothesis that positive results are not just from modeling local systematic error, we trained decision stumps on the data from one location (X) and used them for dispatch in four other areas (A,B,C,D). As shown in Table 8, in three of the four comparisons, the knowledge learned in one area transfers well to the other areas. Note that this is especially true for the Cleaned data. The number of training examples were 5000, 2000, and 3000 for Field, Cleaned, and In vs Out, respectively. The number of test examples varies for each site. All numbers reported are the averages of testing ten decision trees on all of the data from each of the sites A, B, C, and D. Note that we report PDER as well as error rates, due to the differing class distributions among the sites.

6 Cost-effective Dispatch

If the field technicians' resolutions are taken to be reasonably reliable, the previous analysis seems to imply that MAX's performance is poor. We are faced with the issue of analyzing this seemingly poor performance in light of evidence to the contrary. The system has been in use for many years and has not had a negative effect on the operations of the company. One explanation could be that the technicians just do not code the trouble resolutions correctly. However, as the results below show, much of this seeming discrepancy can be explained by the fact that accuracy (or error rate) is not the best metric with which to evaluate dispatch effectiveness.

In the domain of local-loop repair and maintenance, the costs associated with the diagnoses vary substantially. Typically, the cost associated with dispatching a trouble outside of the central office is greater than dispatching to the central office, with the highest cost being associated with dispatching cable technicians. By analyzing the cases for which the decision stump and MAX differ in their dispatches, we find that MAX is making conservative decisions with respect to cost. Thus a convincing comparison of methods for local-loop dispatch must be made with respect to cost-effectiveness in addition to accuracy (Pazzani et al. 1994, Provost 1994). Our focus for this section will be on the three-class version of the MAX problem (cleaned-up data).

6.1 Evaluating Results: Cost-Effectiveness and Accuracy

We now consider the cost that would be incurred by any incorrect decisions made. This task is complicated by the fact that, as discussed in the decision analysis literature (Weinstein & Fineberg 1980), it is often difficult to estimate costs. For instance, certain tests in the central office might require much more time than others, resulting in higher labor costs to determine that

Table 6: Comparison of error rates of Learned Decision Stumps (LDS) and MAX. Standard deviations are given in parentheses, except (*), which indicates that the evaluation was performed on the entire data set, rather than on a small test set reserved after learning.

	Field Data (4 class)	Cleaned (3 class)	In vs. Out (2 class)
MAX	.67 (*)	.79 (*)	.58 (*)
LDS	.52 (.01)	.34 (.01)	.07 (.01)
MAX-D	.67 (.01)	.42 (.03)	.04 (.01)
LDS-D	.52 (.01)	.31 (.02)	.04 (.01)
LDS-C	.34 (.10)	.27 (.02)	.06 (.01)
Default	.62 (.01)	.47 (.01)	.09 (.01)

Table 7: Coverages of test data. Standard deviations are given in parentheses.

	Field Data (4 class)	Cleaned (3 class)	In vs. Out (2 class)
MAX	100 (0.00)	100 (0.00)	100 (0.00)
LDS	100 (0.00)	100 (0.00)	100 (0.00)
MAX-D	99.43 (0.18)	56.59 (1.90)	55.66 (1.78)
LDS-D	99.43 (0.18)	56.59 (1.90)	55.66 (1.78)
LDS-C	9.30 (3.15)	72.19 (5.60)	99.27 (0.52)
Default	100 (0.00)	100 (0.00)	100 (0.00)

the trouble is elsewhere. We interviewed experts to determine, as well as we could, the error costs associated with each of the three dispatchs (PDF, PDO, PDI). Our best approximation is a *cost ratio* of 3:2:1 (PDF:PDO:PDI), with the cost of a central office dispatch (PDI), the *base cost*, about $50.

A naive approach to cost-sensitive classification is to use error costs such as these in combination with estimates of the probabilities of the classes to determine which dispatch will yield the lowest expected cost (EC). The corresponding naive approach to evaluating cost-effectiveness is to classify a test set with the learned classifier and sum up the costs of each incorrect dispatch, using the costs defined above. This is the approach that has been taken in most prior work on cost-sensitivity in the machine learning literature (Turney 1996).

This naive approach is problematic for multi-class problems, because it assumes that after the dispatch is identified as being incorrect, the subsequent dispatch will be correct. The problem can be seen clearly in the following example. Given the 3:2:1 cost ratio defined above, assume that the estimated probability distribution of classes (PDF:PDO:PDI) is 0.5:0.4:0.1. In this case the dispatch with the (naive) minimum expected cost is PDI: $EC(PDI) = .9 * 50 = 45$, $EC(PDO) = .6*100 = 60$, $EC(PDF) = .5*150 = 75$. However, a choice of PDI would be incorrect 90% of the time, and in most cases would not make the choice between PDO and PDF any easier.[7] Indeed, such a naive strategy yields undesirable results in practice.

The alternative is to take a more complex, decision-analytic approach, in which the expected-cost calculation takes subsequent decisions into account. Ideally, for determining the best dispatch for a given trouble, we would like to use the frequencies of classes at the leaves of the decision stump to estimate the class probability distributions for all possible combinations of decisions, in order to calculate the minimum expected-cost dispatch. However, one goal of this analysis is a comparison with the dispatch decisions of the MAX expert system. For MAX, we know only the first dispatch; we do not know what subsequent decisions MAX would make. Thus, using the probability distributions at the leaves of the decision stump for more than just the first decision may give the decision stump an unfair advantage in the comparison, because if MAX were programmed differently, it would be able to issue recommendations for subsequent dispatches as well.

In sum, we are faced with a dilemma; it is obviously important to take subsequent dispatches into account, but we do not know what subsequent dispatches MAX would make. To resolve the dilemma, we used the prior probability distribution of the classes to determine likely subsequent decisions. This information is built into a cost matrix, so that it can be used both

[7] In fact, we assume independence of solutions.

Table 8: Comparison of error rates of knowledge learned from location X when applied to other locations.

Location	Field	PDER	Cleaned	PDER	In vs Out	PDER
X	.52 (.01)	16	.34 (.02)	27	.07 (.01)	29
A	.54 (.01)	19	.25 (.01)	51	.05 (.01)	74
B	.57 (.01)	4	.38 (.01)	25	.07 (.01)	-12
C	.56 (.01)	7	.21 (.01)	58	.03 (.01)	2.3
D	.64 (.01)	-2	.51 (.04)	42	.18 (.01)	-2.2

to evaluate the decisions of classifiers (such as MAX) that give only a single answer, and to choose cost-sensitive dispatches in cost-sensitive classifiers. Fortunately, we found that there is very little difference in cost-effectiveness between using the prior probability distribution and the leaf probability distribution for determining the second dispatch when the first is wrong. We will now describe in detail the process of building cost matrices that take subsequent (expected) errors into account.

First let us define the function $cost(x)$, which, based on the cost vector, gives the cost of mistakenly choosing dispatch x.

For the *naive approach*, the cost matrix is built by assigning

$$NCost(p)(a) = \begin{cases} cost(p) & \text{if } p \neq a \\ 0 & \text{otherwise} \end{cases}$$

where p = *predicted* and a = *actual*. For the *decision-analytic approach*, we assume that the subsequent dispatch will be the minimum expected-cost dispatch of the remaining choices, based on the prior probability distribution. Suppose there are three classes, X, Y, and Z, and let $p = X$.

$$DACost(p)(a) = \begin{cases} cost(X) + SecCost & \text{if } a \neq X \\ 0 & \text{otherwise} \end{cases}$$

Without loss of generality, let $a = Y$, then

$$SecCost = \begin{cases} cost(Z) & \text{if Z is the min exp-cost class} \\ & \text{between Y and Z} \\ 0 & \text{otherwise} \end{cases}$$

In this case, the expected cost of a secondary dispatch, e.g., Y, is the probability of Y being wrong times the cost of being wrong,[8] or $(1 - (p(Y)/(p(Y) + p(Z)))) * cost(Y)$, where $p(Y)$ and $p(Z)$ are the prior probabilities. The fractional probability term is due to the removal of X as a possible correct *secondary* dispatch.

An example of a decision-analytic cost matrix calculated from example costs and data priors is given in Table 9. Note that all costs here are error costs. The cost is zero for correct dispatches.

[8]When Y is correct, the error cost is zero.

6.2 Building a Cost-sensitive Decision Stump

We built cost-sensitive decision stumps by recording at each leaf a frequency-based probability estimate for each class. The estimate was calculated as (TP/(TP+FP)), where TP is the true-positive coverage of the leaf and FP is the false-positive coverage of the leaf. When the cost-sensitive stump is used, it uses the conditional probabilities at the leaves to dispatch to the minimum expected-cost class, using the decision-analytic cost matrix (built using prior probabilities from the training data to determine expected subsequent dispatches, as described above). Note that Pazzani et al. (1994) found that estimating class probabilities at the leaves of a decision tree and using these for a minimum expected-cost calculation is not effective at reducing cost; they account for this phenomenon by noting that the probability estimates at the leaves of a decision tree are based on small samples, and thus are inaccurate. Since we use a decision stump, we hope that the larger numbers of examples at the leaves will lead to better probability estimates.[9]

Results comparing MAX with the vercode decision stump and cost-sensitive decision stump are summarized in Table 10. The cost matrix used to generate these results is that in Table 9. A simplistic comparison of the performance of MAX, the vercode stump, and the cost-sensitive stump (first, second, and fifth rows of the table) shows that although the dispatches made by the vercode decision stump are more accurate than those of MAX, the decisions made by MAX are more cost-effective. The cost-sensitive decision stump reduces the cost without losing accuracy.

However, this comparison masks an important subtlety. Specifically, as with the earlier error-rate comparisons, MAX only gives a dispatch recommendation on (approximately) 57% of the cases; the rest are routed for further testing or for human analysis. On other hand, the stumps give dispatch recommen-

[9]Recent work suggests that cost-sensitive classification with decision trees can be quite effective, if the probabilities are generated using the Laplace estimate rather than a simple frequency-based estimate (Bradford et al. 1998). The Laplace estimate protects against unwarranted optimism due to small samples.

Table 9: Cost matrix for dispatch classes in the MAX domain. Rows are predicted classes. Columns are actual classes. Classes:(PDF:PDO:PDI) Costs:(150:100:50) Priors:(0.51:0.35:0.14)

	PDF	PDO	PDI
PDF	0	150	250
PDO	100	0	250
PDI	50	200	0

dations on 100% of the cases.

In Table 10, we therefore also report the error rate (ER) and Error Cost per Dispatch for the stumps on those cases for which MAX gave a dispatch recommendation (MAX-D), and on those cases for which the stumps were confident of their recommendation (i.e., the probability of class membership was ≥ 0.6).

As expected, the decision stumps perform considerably better on both subsets of cases, in terms of both error rate and cost. Perhaps surprisingly, the difference in performance between the cost-sensitive and non-cost-sensitive stumps is no longer apparent when they are evaluated on the subsets. This is because as the required confidence level is raised, the behaviors of the two types of stump are more and more similar, eventually becoming identical. Apparently, a threshold of 0.6 is sufficient (effectively) for the cost matrix being used.

6.3 Sensitivity Analysis

While the results above suggest that it is possible to learn cost-sensitive decision stumps that are both more accurate and more cost-effective than MAX, we must have confidence that this is not due to a fortuitous choice of costs (especially since the specification of costs is far from perfect). To this end, we perform an analysis of the evaluation's sensitivity to changes in the cost ratio.

For this paper, we consider varying only the ratio PDF:PDO, holding the ratio PDO:PDI at 2:1. Consider the cost ratio to be X:1:0.5 (PDF:PDO:PDI). Figure 1 shows the effect of varying X from 1 to 3 in increments of 0.1 on the error costs associated with the dispatches made by MAX, the decision-stump, and the cost-sensitive decision stump, using decision-analytic cost matrices constructed as described above. Figure 2 and Figure 3 show the effects of varying X for the stumps when evaluated on MAX-dispatched and confident cases, respectively.

As would be expected, the cost per dispatch of the decision stump increases smoothly (and linearly) with the increasing cost of making PDF errors. The decision stump always makes approximately the same percentage of PDF errors, so as the cost of a PDF error increases linearly, so will the cost-per-dispatch of the decision stump.

The performance of MAX as the cost of a PDF error increases is more interesting. Inspection reveals that the curve representing MAX's error cost per dispatch is (approximately) piecewise linear with increasing PDF error cost, and the slope of each segment is less than the slope of the decision stump curve. The relatively low slope of each segment is due to the fact that MAX errs on the conservative side; specifically, it makes fewer PDF errors than the decision stump. Thus, the growth of the overall cost per dispatch as the PDF error cost grows will be smaller.

The discontinuity when the PDF:PDO error cost ratio reaches 2:1 can be explained by examining the changes in the cost matrices as the ratio increases. In particular, consider the two cost matrix entries $DACost(PDF)(PDO)$ and $DACost(PDO)(PDF)$.[10] Across the range of ratios represented in the graph, $DACost(PDF)(PDO) = cost(PDF)$, because in this range PDO is always the minimum expected-cost secondary dispatch. Similarly, when the ratio of the error cost of PDF to PDO is in the range $[1, 2)$, $DACost(PDO)(PDF) = cost(PDO)$. This explains technically why the slope of the MAX curve is less: $cost(PDO)$ is constant; $cost(PDF)$ increases linearly. However, when the ratio is in the range $[2, 3]$, $DACost(PDO)(PDF) = cost(PDO) + cost(PDI)$, because due to the prior distribution of classes, PDI becomes the secondary dispatch of choice. Thus $DACost(PDO)(PDF)$ is still constant, but it is greater than it was over the prior range. Hence the curve's piecewise linearity. This reasoning applies to the vercode stump as well, though the difference in slopes of the two line segments is very slight, and thus difficult to recognize from the graph.

The result of these two factors is that MAX's performance, in terms of error cost per dispatch, is better than the decision stump when the ratio of the cost of a PDF error to a PDO error is in the range $(1.4, 2)$. As mentioned above, our problem analysis determined (independently) that the actual cost ratio is approximately 1.5. Thus, the design of MAX and the years of tuning its performance in the field seem to have been effective.

[10] Recall that the cost matrix entries are of the form $DACost(predicted)(actual)$.

Table 10: Comparison of Vercode Decision Stumps and MAX. Training sets of 2000 examples used to build the Vercode stump and Cost-sensitive stump (ER = error rate). Independent test sets of 686 examples used to test. All avgs are over 10 runs. Standard deviations are given in parentheses.

	Total Preds Made	ER	Error Cost per Dispatch
MAX (MAX-D)	56.69% (1.9)	0.42 (.03)	50.61 (3.87)
Vercode stump	100% (0.0)	0.34 (.01)	51.44 (2.35)
Vercode stump (MAX-D)	56.69% (1.9)	0.31 (.02)	43.77 (2.66)
Vercode stump (Conf)	72.19% (5.6)	0.27 (.02)	39.41 (3.00)
Cost-sensitive stump	100% (0.0)	0.35 (.02)	47.50 (2.47)
Cost-sensitive (MAX-D)	56.59% (1.9)	0.33 (.01)	43.78 (2.58)
Cost-sensitive (Conf)	70.90% (5.1)	0.27 (.02)	39.08 (2.75)
Always dispatch PDF	100% (0.0)	0.47 (.01)	83.92 (2.41)
Always dispatch PDO	100% (0.0)	0.66 (.01)	85.93 (2.15)
Always dispatch PDI	100% (0.0)	0.87 (.01)	94.88 (2.67)

The graph also shows that the cost-sensitive stump adjusts for the different cost ratios automatically. Across the entire range, the cost-sensitive stump outperforms the other two methods, although the difference is small in the range of MAX's maximal effectiveness. Statistical tests on individual points do not indicate that the difference is statistically significant, and statistical tests on the entire curves are difficult because of interdependencies in the generation of the different points. However, it is not clear that statistical significance is particularly important. Even if the performance were indistinguishable, the cost-sensitive decision stumps are preferable for their simplicity, for their flexibility in adapting to different cost scenarios, and for their ease of updating. From the perspective of business significance, a potential cost savings of two or three dollars per dispatch is very significant. Also, it should be noted that (as above) these comparisons are somewhat unfair to the decision stumps, because they are making recommendations on all cases, whereas MAX is only making recommendations on about half of the cases.

Figure 2 shows that both the vercode decision stump and the cost-sensitive decision stump outperform MAX across the entire range, when classifying only those cases on which MAX makes a recommendation. Again, as would be expected, the cost per dispatch of the decision stump increases smoothly and linearly with the increasing cost of making PDF errors. It simply makes fewer of these errors when asked to make a call on fewer cases.

Also, as expected, the vercode decision stump and the cost-sensitive stump perform similarly in the range [1,2), where the cost ratio PDF:PDO is relatively low. When the PDF cost becomes increasingly large, the cost-sensitive stump adjusts for that cost, diverting classifications to less expensive dispatches.

An even greater disparity in error costs is seen when the vercode stump and cost-sensitive stump dispatch only on confident cases.

7 Summary and Discussion

For this case study we have selected a series of experiments that highlight the variety of perspectives that must be taken in order to determine the potential for inductive learning programs to be applied successfully. The case study highlights issues of problem definition, data cleaning, and evaluation that are usually glossed over (or simply ignored) in most published reports on classifier learning. Taken in total, the results provide solid evidence that simple inductive learning programs can learn effective classifiers for local-loop troubleshooting.

At first glance, the primary result is that decision stumps can be learned that are more accurate and more cost-effective than the troubleshooting system currently in place. What is more interesting, however, is that the stumps achieve at least equivalent performance with much less effort in design, implementation, and tuning. This suggests that dealing with new equipment or with different local environments (e.g., Manhattan versus Maine) will be much easier. In the long run, being able to do a better job of keeping systems well-tuned may magnify the differences in performance observed here.

From the standpoint of the machine learning and the knowledge discovery communities, the study is most interesting as a counterbalance to the prevailing narrow view of classifier learning. In the first place, in most inductive learning research the correctness of the class labels is a basic assumption that goes unquestioned. Perhaps more strikingly, although it is difficult to imagine a real-world problem for which all errors

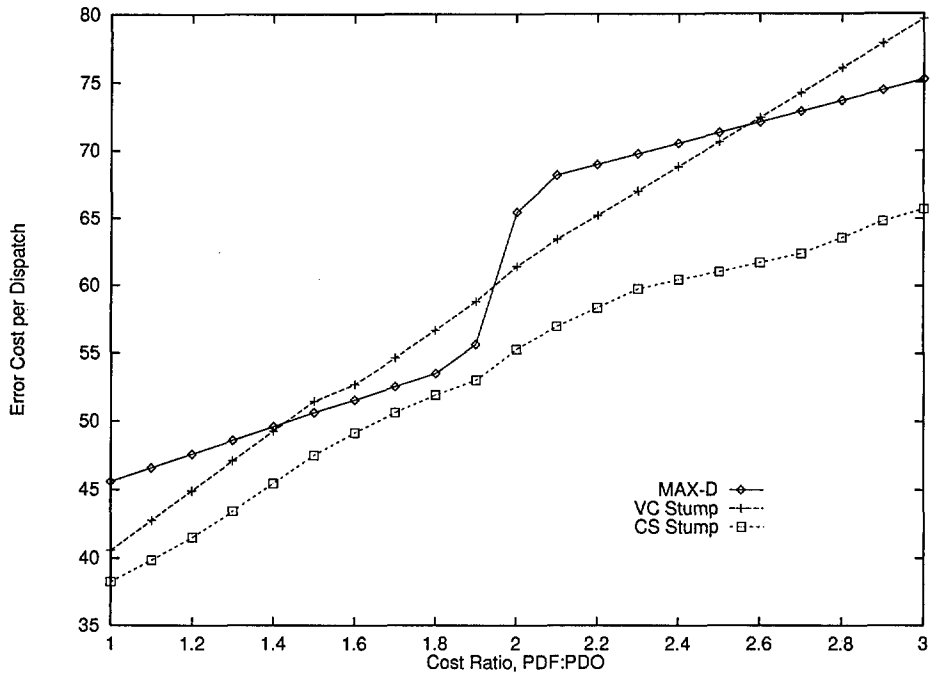


Figure 1: Effect of Varying Error Cost Ratios on Cost of Errors Made

have equal costs, equal error cost is another unquestioned assumption of the vast majority of research on inductive learning.[11] This case study shows how each of these assumptions can lead to a misleading evaluation.

7.1 Lessons Learned

From this case study we can draw several general lessons that we believe are applicable to many real-world machine learning and data mining applications. We have found some to echo lessons learned in other applications work (Kohavi & Provost 1998).

Lesson 1: A single source of data gives a narrow view of the problem. In order to get the complete, well-rounded picture necessary to present a compelling argument for the real-world use of this technology, we found it necessary to use multiple data sources and to perform data cleaning based on domain knowledge.

Lesson 2: Superficial use of accuracy figures gives a shallow view of the problem. Be careful not to fall into the trap of ignoring the accuracy of simple methods. All too often the inexperienced data miner is elated by the seemingly good performance of his (or her) favorite learning program, only to discover that a simple method (e.g., a linear discriminant function or simple Bayes) works just as well, or more embarrassingly, that the class distribution is highly skewed. Hopefully, such a discovery is made before the results are presented to someone for whom retraction would be an embarrassment. This paper shows a case where a simple method (decision stump) actually outperforms a more complex decision-tree learner (as well as other complex learning programs).

Also, one should be careful not to compare incomparable accuracy figures. The most obvious reason why inter-study accuracy comparisons would not be valid is that the different data sets have different class distributions. We saw evidence of this in the section on data cleaning; the different class distributions were due to the fact that cleaning the data both eliminated classes from the data and removed troubles non-uniformly across the remaining classes. For our inter-study comparisons, we used the percentage decrease in error rate for each metric over the error rate of the default class.

Lesson 3: Broad comparison studies increase the confidence in the evaluation. When arguing for the use of inductive learning technology, the last thing you want is to be blindsided by questions like "How does it compare with the (currently used) FooBar system," or "Well ... Brooklyn is a special case, have you tried data from Upstate?" We were lucky to have considerable management and peer support and enthusiasm, which is certainly not universal in real-world applications of emerging technologies. In addition to the use of multiple data bases, and multiple learning methods described in Lessons 1 and 2, we also found it necessary to produce multiple "existing" methods with which to compare, including the existing expert system, as well as the experts themselves. Furthermore, we found it necessary to collect data from geographically disparate

[11] For a detailed analysis of this assumption, see the recent paper by Provost, Fawcett and Kohavi (Provost, Fawcett, & Kohavi 1998).

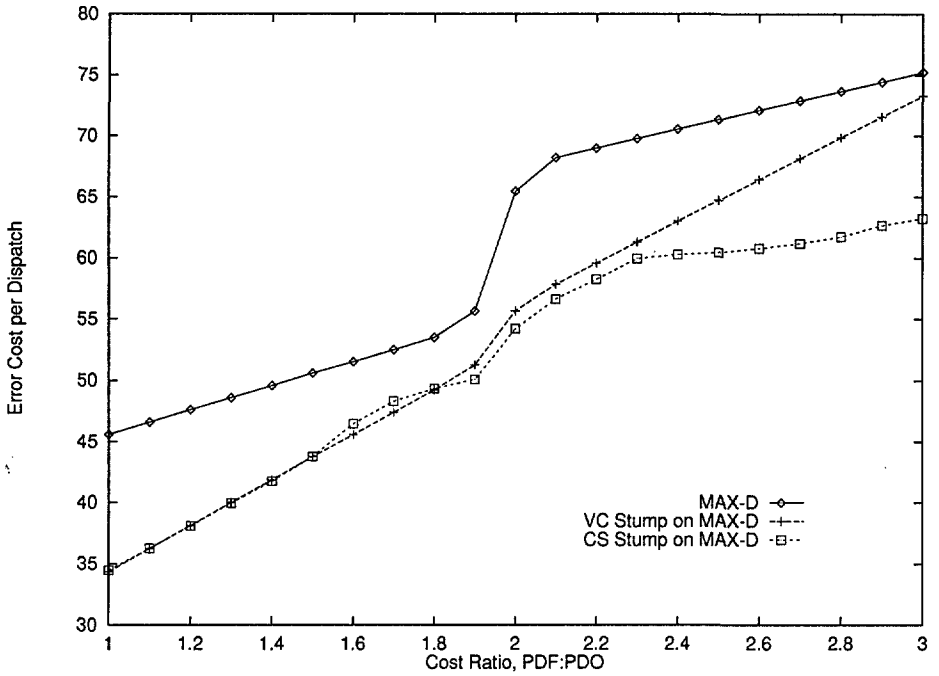


Figure 2: Effect of Varying Error Cost Ratios on Cost of Errors Made. Data are restricted to MAX-dispatched cases.

locations to demonstrate the robustness of the learning.

Lesson 4: Don't lose sight of the real performance task. In real-world domains, accuracy is seldom the bottom line. More often, cost-effectiveness is. In many domains, different errors have different associated costs, so it is important that the learned knowledge produce the right trade-offs. In this domain, not only did we find that an analysis of the cost-effectiveness of the learned classifiers is essential, we also discovered that merely paying lip service to cost-effectiveness with a naive cost analysis is not sufficient. We had to bring in techniques from decision analysis (that are seldom even mentioned in machine learning/data mining research—more below).

7.2 Implications for Inductive Learning Research

We believe that studies such as this of the actual use of inductive learning, in a practical application where there is high-level support for the use of AI technologies (and therefore complexity does not come from a distrust of the techniques), should be a guiding influence to the research community. Therefore, let us discuss briefly the type of research results that would have been helpful to this effort.

Dealing with potentially erroneous data was a major issue. Most of the machine learning/data mining literature on learning in the presence of noise discusses random errors. However, the types of errors most often discussed in this domain (and many real-world domains) usually have some degree of systematicity—systematicity that may also appear in the evaluation data. Furthermore, we have not performed a detailed analysis of the effect of data cleaning. For example, we eliminated from the data borderline cases and cases for which we could not determine the correct resolution. How will this affect our evaluation? We believe that learning in the presence of possible systematic errors is a very interesting open problem (Weiss 1995, Beers 1957, Lee 1995).

The machine learning community has spent the last decade comparing learning programs on suites of benchmark problems *ad nauseam*. However, almost all evaluations have been based on classification accuracy, the result being a host of available systems that can maximize accuracy within their inductive biases. We believe that unless the accuracy is 100%, very few real-world domains use classification accuracy as the prime evaluation criterion. In fact, evaluations based on classification accuracy can be quite misleading (Saitta & Neri 1998, Provost et al. 1998).

When we were faced with the prospect of learning with sensitivity to the cost of errors, we found ourselves with only a handful of small-scale, comparatively inconclusive studies in the machine learning literature. We believe it is time for machine learning and data mining research to take off the blinders of classification accuracy and develop robust methods that can provide the cost-effective classification needed in the real world. Interested researchers can begin by referencing work in statistics (Duda & Hart 1973),

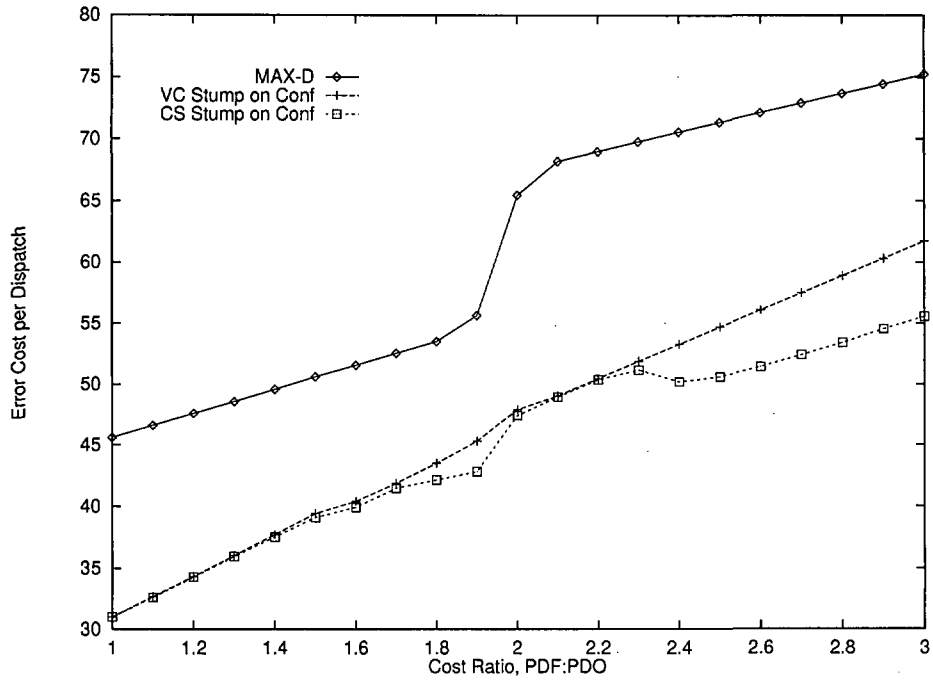


Figure 3: Effect of Varying Error Cost Ratios on Cost of Errors Made. Data are restricted to confidently dispatched cases.

decision analysis (Henrion et al. 1991, Keeney 1982, Weinstein & Fineberg 1980), and pattern recognition (Dattatreya & Kanal 1985). Also, Turney (1996) provides an on-line bibliography of work on cost-sensitive machine learning.

Acknowledgements

Tom Fawcett helped in developing new mappings from field technicians' resolution codes to dispatches. Kim Tabtiang performed the runs reported in Sections 3, 4, and 5. Peter Turney and Irene Po initiated the discussion of using a decision-analytic cost matrix, and built the first prototype. We would like to thank the following colleagues for valuable discussions over the course of the Machine Learning for MAX project: Charlie Bove, Bruce Buchanan, Jim Euchner, Tom Fawcett, Neil Melley, Mike Pazzani, Irene Po, Henry Rabinowitz, Jude Shavlik, Judy Spitz, Peter Turney, Rikki Wolin, and Yuling Wu.

References

[1] Beers Y. (1957) *An Introduction to the Theory of Error.* Addison-Wesley Publishing Company, Reading, MA.

[2] Bradford J., Kunz C., Kohavi R., Brunk C. & Brodley C. (1998) Pruning decision trees with misclassification costs. *Proceedings of ECML-98*, p. 131-136.

[3] Danyluk A. P. & Provost F. J. (1993a) Small Disjuncts in Action: Learning to Diagnose Errors in the Local Loop of the Telephone Network. *Proceedings of the Tenth International Conference on Machine Learning*, p. 81-88. Morgan Kaufmann Publishers, Inc., San Mateo, CA.

[4] Danyluk A. P. & Provost F. J. (1993b) Adaptive Expert Systems: Applying Machine Learning to NYNEX MAX. *Working Notes of the AAAI-93 Workshop on AI in Service and Support: Bridging the Gap Between Research and Applications*, Washington, DC, p. 50-58.

[5] Dattatreya G. R. & Kanal L. N. (1985) Decision Trees in Pattern Recognition. In L. N. Kanal and A. Rosenfeld (ed.), *Progress in Pattern Recognition 2*, p. 189-239. Elsevier Science Publishers B.V. (North-Holland).

[6] Duda R. O. & Hart P. E. (1973) *Pattern Classification and Scene Analysis.* John Wiley, New York.

[7] Henrion M., Breese J. S. & Horowitz E. J. (1991) Decision Analysis and Expert Systems. *AI Magazine*, 12, p. 64-91.

[8] Holte R. C. (1993) Very Simple Classification Rules Perform Well on Most Commonly Used Datasets, *Machine Learning*, 11, 1, p. 63-90.

[9] Keeney R. L. (1982) Decision Analysis: An Overview. *Operations Research*, 30, p. 803-838.

[10] Kohavi R. & Provost F. (1998) *Special Issue on Applications of Machine Learning and the Knowledge Discovery Process*, Machine Learning, 30, 2/3.

[11] Lee Y. (1995) *Learning a Robust Rule Set.* Ph.D. Thesis. Department of Computer Science, University of Pittsburgh.

[12] Merz C. J., Pazzani M. & Danyluk A. P. (1996) Tuning Numeric Parameters to Troubleshoot a Telephone Network Local Loop. *IEEE Expert*, 11, 1, p. 44-49.

[13] Pazzani M. J. & Brunk C. (1993) Finding Accurate Frontiers: A Knowledge-Intensive Approach to Relational Learning. *Proceedings of AAAI-93*, p. 328-334, AAAI Press, Menlo Park, CA.

[14] Pazzani M., Merz C., Murphy P., Ali K., Hume T. & Brunk C. (1994) Reducing Misclassification Costs. *Machine Learning: Proceedings of the Eleventh International Conference*, p. 217-225. Morgan Kaufmann, San Mateo, CA.

[15] Provost F. (1994) Goal-Directed Inductive Learning: Trading off Accuracy for Reduced Error Cost. *Working Notes of the AAAI-94 Workshop on Goal-Driven Learning*, p. 94-101.

[16] Provost F. & Aronis J. (1996) Scaling Up Inductive Learning with Massive Parallelism. *Machine Learning*, 23, p. 33-46.

[17] Provost F., Fawcett T. & Kohavi, R. (1998) The Case Against Accuracy Estimation for Comparing Induction Algorithms. *Machine Learning: Proceedings of the Fifteenth International Conference*, p. 445-453. Morgan Kaufmann Publishers, San Francisco, CA.

[18] Quinlan J. R. (1993) *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, Inc., San Mateo, CA.

[19] Rabinowitz H., Flamholz J., Wolin E. & Euchner J. (1991) NYNEX MAX: A Telephone Trouble Screening Expert. In R. Smith and C. Scott (ed.), *Innovative Applications of AI 3*, p. 213-230, AAAI Press, Menlo Park, CA.

[20] Saitta L. & Neri F. (1998) Learning in the "Real World", *Machine Learning*, 30, 133-164.

[21] Turney P. (1996) Cost-sensitive learning bibliography.

http://ai.iit.nrc.ca/bibliographies/cost-sensitive.html.

[22] Weinstein M. C. & Fineberg H. V. (1980) *Clinical Decision Analysis.* W.B. Saunders Company, Philadelphia, PA.

[23] Weiss G. M. (1995) Learning with Rare Cases and Small Disjuncts. *Proceedings of the Twelfth International Conference on Machine Learning*, p. 558-565, Morgan Kaufmann, San Francisco, CA.

Research on Telework in Slovenia

Eva Jereb, Miro Gradišar
Faculty of Organisational Sciences, University of Maribor
Kidričeva 55a, 4000 Kranj, Slovenia

Keywords: telework, teleworkers, office automation

Edited by: Rudi Murn
Received: April 24, 1998 **Revised:** September 16, 1998 **Accepted:** December 9, 1998

Although a lot of research on telework has been done, most of it has been carried out in USA and EC countries. Until today there have been no studies on telework in Slovenia. This paper presents the results of a study of telework in Slovenia carried out in 1997. The paper also compares telework between Slovenia and other European countries.

1 Introduction

Information and communication technology provides organisations with a new flexibility as regards where, when and how work is performed, which gives rise to a number of new organisational forms and new ways of performing work [16,19]. Through the implementation of electronic information systems, the structure, procedures and content of office work are changing. Increased attention is now being paid to a new form of work known as telework. Telework allows spatially and organisationally decentralised office work, with work results being sent back electronically via communication networks. This has only become possible in the last few years through the developments in the areas of distributed information systems, office aut omation and telecommunications [13].

In the 70's some authors had high expectations when they believed that "all Americans could be homeworkers by 1990" [11]. A decade later others stated that "by the year 2000 approximately 40% of the employees in the US will be teleworkers". The latest estimate is that in many development countries 10%-15% of the workforce will be teleworking to some degree by the end of the century [15].

In 1990 there were 2 million teleworkers in US, in the year 1994 7.8 million and the estimations in the year 1994 were that in 2001 there will be 30 million teleworkers [2].

Different surveys [1,6] have shown that telework is slowly penetrating into Euro pe as well. According to the results of surveys in 1994 the number of teleworkers in the five largest European Community countries was approximately 1.1 million by the time. Extrapolating this figure to the whole Europe gives a total of 1.25 million tel eworkers. In terms of absolute numbers of teleworkers, the United Kingdom had the most with 560 000, followed by France with around 215 000 teleworkers, Germany with 149 000, Spain with 102 000 and Italy with 97 000 teleworkers.

Telework is being introduced slowly step by step. In most cases, first the combination of working in the company and working at home is being practised. At first teleworkers are working at home for an average of 4.2 days per month, after a year or two for about 8 days per month [3,17,20].

Because of the benefits the interest in telework is growing among employers and employees. Benefits to employees are: saving on time, money and effort in commuting to work, better concentration at work, flexible working hours, better balance between work and family life. Telework enables taking care of young children, elderly or disabled relatives and allows into employment people unable to work in the traditional way, such as disabled or handicapped people. The main advantages to employers are: increased productivity, better office space utilisation, reduced overhead expenses, reduced travel costs, reduced electricity, food and other costs [9,10,18].

Of course telework has its drawbacks as well. Employers are concerned about data security and loss of control. Some are concerned about the legal rights and normal protection in law that employees are afforded [7]. Telework may increase the cost of living for the teleworker (home office heating and power, food, ...) [8]. Teleworkers may not be keen on carrying out their own typing, filling, and other routine office functions. Some teleworkers may miss the social interaction of the workplace. The feeling of belonging to a team that is working for a common goal may be lost [4].

In the paper we discuss the methodology used in our research, the instrument, data collecting and results. In the end we show some results of the Empirica survey [12] carried out in 1990 in 14 European companies parallel with results of our survey carried out in 1997 in 15 Slovenian organisations and give an conclusion.

	Need for flexible working hours				Worry for promotion and career			Importance of personnel contacts		
	High	Medium	Low	No need	Yes	No	Don't know	High	Medium	Low
People, who are interested in telework	25%	43%	19%	13%	14%	42%	44%	57%	40%	3%
People, who are not interested in telework	13%	29%	33%	25%	21%	17%	62%	83%	13%	4%

	Saving on effort in commuting				Loss of review of what's going on in company			
	Very high	High	Medium	Low	Completely	Partly	No	Don't know
People, who are interested in telework	6%	16%	46%	32%	10%	72%	15%	3%
People who are not interested in telework	0%	8%	29%	65%	42%	45%	0%	13%

Table 2: Telework affected by psychological and sociological factors

When the results are analysed according to company size, next trend comes to light: the greater the size of a company, the more interest is in telework. That holds for France, Germany and United Kingdom but not for Italy [12].

We wanted to find out how company size affects telework in Slovenia. We predicted that company size does not affect interest in teleworking and test showed that on the risk level 0.05 we can accept our prediction.

In the end we show some results of the Empirica survey [12] carried out in 1990 in 14 European companies parallel with results of our survey carried out in 1997 in 15 Slovenian organisations.

3 Results of teleworking surveys in different European countries

One of the questions we were interested in was "why were companies interested in introducing telework". This is of course a question with two sides, that of the employer and the employee, since telework can only take place when there is sufficient convergence of interest for both parties to agree to it.

We turned to the employer's side. There is in fact a wide range of reasons companies might have for considering the introduction of remote work. We identified six common motivations: increased productivity, reduced commuting costs, reduced central office's costs, flexibility in working hours, employment of the disabled and retention of scarce skills. In Table 3 we are showing the importance of these reasons to managers in Slovenia in 1997 and to managers in other European countries in 1990.

As it can be seen from the table these reasons are very important to managers from Slovenian companies. It is hard to say why but it may be due to the fact that Slovenia is a small country with a population of 2 million which became independent in the year 1991. In the transition to a market economy and private ownership, most larger companies have disintegrated and managers of these smaller companies want to make a good use of the telework possibilities in order to gain bigger competitive advantage. Increased productivity and flexibility of working hours are for our managers the most important reasons for introducing telework.

Communication is a particularly important component of distance working, so the employees were asked about the use of communication media. Table 4 summarises the results.

We can see that employees in Slovenia spend a lot of time on telephoning and meetings. What I think might be worth considering.

Table 5 shows the proportion of work done at home or outside normal working hours by gender. We can see that the majority do perform some of their work in the evenings or at weekends. That goes for Slovenian and other European companies. In Slovenia there are more men then women who do a part of their work at home or outside their normal working hours. In other European countries in the 1990 there were more women then men.

In table 6 we are showing interest in telework of men and women in different European countries.

As we can see employees in Slovenian companies show very high interest in telework. Slovenia has not been independent for a long time and maybe employees are now looking for new chances to succeed.

	Slovenian companies (1997)			Other European companies (1990)		
	Important (%)	Unimportant and no answer (%)	Rank order	Important (%)	Unimportant and no answer (%)	Rank order
Increased productivity	87	13	1	36	64	2
Reduced commuting costs	71	29	4	29	71	3
Reduced central office's costs	75	25	3	29	71	3
Flexibility in working hours	83	17	2	29	71	3
Employment of disabled	71	29	4	14	86	4
Retention of scarce skills	52	48	5	43	57	1

Table 3: Companies' reason for introducing telework

		No use (%)	Once a week (%)	Several times per week (%)	Once a day (%)	Several times a day (%)
	Slovenia 1997	7	1	5	2	85
Telephone	Europe 1990	46	3	19	8	24
	Slovenia 1997	93	3	3	0	1
Teletex	Europe 1990	98	2	0	0	0
	Slovenia 1997	98	2	0	0	0
Videotex	Europe 1990	99	1	0	0	0
	Slovenia 1997	60	7	14	4	15
E-mail	Europe 1990	84	3	6	5	2
	Slovenia 1997	37	27	17	8	11
Postal service	Europe 1990	29	7	47	11	6
	Slovenia 1997	62	17	11	6	4
Courier	Europe 1990	69	21	7	2	1
	Slovenia 1997	18	2	15	5	60
Meetings	Europe 1990	25	45	27	3	0

Table 4: Employees' use of communications media

4 Conclusion

There are only few companies already practising telework, but a lot of them are thinking about introducing it. Although the US is currently the leader in this process, interest in the rest of the world, particularly Europe, is accelerating. Interest in telework is seen also in Slovenia. In the empirical research among Slovene organisations carried out in summer 1997, we found out that managers and employees are interested in telework. We found out that technology needed for telework is not the basic problem in introducing telework and that almost all of potential teleworkers have their own for teleworking needed equipment. The survey also showed that technological factors and the content and way of work performed within a specific working place determine its suitability for telework and that telework is strongly affected by psychological and sociological factors. We also found out that managers are sometimes troubled by the idea facing the prospect of managing a team of remote workers and they know that their tasks will partly change and that also the way of control and way of policy making will change. Managers are also concerned about employment contracts, most of them think that it would be necessary to legally define the working conditions of teleworkers and clearly state the unique responsibilities of both parts.

References

[1] Action for Stimulation of Transborder Telework and Research Activities in Europe, Telework 1995, European Commission, DG XIII-B, `http://www.acad.by/wise/english/rd/-reports/telework/annual95`

[2] Brokaw T.: NBC Nightly News, March 22, 1994, `http://www.teleworker.com/quotes.html`

[3] City of Los Angeles Telecommuting Project: Final Report, JALA International, Los Angeles, California, March 1993, pp.26, `http://www.teleworker.com/quotes.html`

[4] Cook E.: The Teleworking Directory Enquiry Experiment at Inverness, Telework Conference,

			None	Up to 1/3	Half	At least 4/3
		n	24	31	0	0
	Slovenia 1997	%	44	56	0	0
		n	18	53	12	2
Female	Europe 1990	%	21	63	14	2
Male		n	34	85	8	8
	Slovenia 1997	%	25	63	6	6
		n	11	7	10	3
	Europe 1990	%	35	23	32	10

Table 5: Proportion of work done at home or outside normal working hours by gender

	Interested (%)		Not interested or not possible (%)		Do not know (%)	
	Male	Female	Male	Female	Male	Female
Slovenia 1997	83	78	10	18	7	4
Germany 1990	9	8	88	87	3	5
France 1990	13	15	84	82	3	3
United Kingdom 1990	22	23	75	72	3	5
Italy 1990	12	10	87	89	1	1

Table 6: Employees interest in telework by gender in European countries

1995, http://www.humanities.mcmaster.cal?misc2/telwtoc.htm

[5] Dekleva S. And Zupančič J.: Key issues in information systems management: a Delphi study in Slovenia, Information & Management 31 (1996) 1-11, 1996 Elsevier Science B.V.

[6] Executive Summary of the TELDET Project Final Report, http://www.acad.by/wise/-english/rd/reports/telework

[7] Flexibility & Exploitation, Telework Conference, http://www.humanities. mcmaster.cal?misc2/telwtel.htm

[8] Gray M., Hodson N., Gordon G.,: Teleworking Explained, John Wiley & Sons, England 1993.

[9] Harper K.: Study shows Home Working offers big gains for companies, Guardian, Jun 1992.

[10] Hendricks B.: Telearbeit - B. unter dem Arm, Wirtschaftswoche Nr.42, Oktober 1993, pp. 123-125.

[11] Huws U.: The new homeworkers. New technology and the changing location of white collar work, Nottingham 1984.

[12] Huws U., Korte W.B., Robinson S.: Telework, Towrds the Elusive Office, John Wiley & Sons, England 1990.

[13] Jereb E. and Jereb J.: Personnel selection for telework by an expert system, Zbornik radova, 8th International simposium of Information Systems IS '97, Varaždin, 24-26 September 1997, pp. 315-326.

[14] Jereb. J and Jereb. E: Introducing the software package OCP Office, IS'95 - 6th International Symposium of Information Systems, September 20 -22, Varaždin, Croatia, F5-F13.

[15] Korte W.B., Kordey N., Robinson S.: Penetration, potential and practice-has it lived up to expectations?, 1995, http://www.acad.by/wise/english/rd/-report/telework/annual95

[16] Lindström J., Moberg A., Rapp B.: On the classification of telework, European Journal of Information Systems (1997) 6, pp. 243-255.

[17] Neitzel W.: Nur einmal pro Woche ins Büro - keine ferne Vision, Die Welt, 19 Julij 1991.

[18] Nilles J.M.: Managing Telework: Strategies for Managing the Virtual Workforce, John Wiley & Sons, England, 1998.

[19] OECD Main Economic Industries, November 1997, http://www.sigov.si/cgi-bin/spl/-zmar/arhiv/og1197/eosl1197.html-?language=3Dslo

[20] Wynne R., Korte W.B., Wynne N.: Telework: Penetration, Potential and Practice in Europe, IOS Press, England, 1995.

Call for Paper
International Multi-Conference
Information Society - IS'99

12 – 14 October, 1999
Slovenian Science Festival
Cankarjev dom, Ljubljana, Slovenia

Programme committee:

dr. Cene Bavec, chairperson,
prof. dr. Ivan Bratko, co-chair,
prof. dr. Matjaž Gams, co-chair,
prof. dr. Tadej Bajd,
mag. Jaroslav Berce,
dr. Dušan Caf,
prof. dr. Saša Divjak,
dr. Tomaž Erjavec,
prof. dr. Nikola Guid,
prof. dr. Borka Jerman Blažič Džonova,
doc. dr. Gorazd Kandus,
doc. dr. Marjan Krisper,
mag. Andrej Kuščer,
prof. dr. Jadran Lenarčič,
dr. Franc Novak,
prof. dr. Marjan Pivka,
prof. dr. Vladislav Rajkovič,
prof. dr. Ivan Rozman,
dr. Niko Schlamberger,
prof. dr. Franc Solina,
prof. dr. Stanko Strmčnik,
prof. dr. Jurij Tasič,
prof. dr. Andrej Ule,
dr. Tanja Urbančič,
prof. dr. Baldomir Zajc,
dr. Blaž Zupan

Invitation

You are kindly invited to participate in the "New Information Society - (IS'99)" multi-conference to be held in Ljubljana, Slovenia, Europe, from 12–14 October, 1999. The multi-conference will consist of seven carefully selected conferences.

Basic information

The concepts of information society, information era, infosphere and infostress have by now been widely accepted. But, what does it really mean for societies, sciences, technology, education, governments, our lives? What are current and future trends? How should we adopt and change to succeed in the new world?

IS'99 will serve as a forum for the world-wide and national community to explore further directions, business opportunities, governmental European and American policies. The main objective is the exchange of ideas and developing visions for the future of information society. IS'99 is a standard scientific conference covering major recent achievements. Besides, it will provide maximum exchange of ideas in discussions, and concrete proposals in final reports of each conference.

The multi-conference will he held in Slovenia, a small European country bordering Italy and Austria. It is a land of thousand natural beauties from the Adriatic sea to high mountains. In addition, its Central European position enables visits to most European countries in a radius of just a few hours drive by car. The social programme will include trips by desire and organised trips to Skocjan or Postojna caves. Coffee breaks, the conference cocktail and dinner will contribute to a nice working atmosphere.

Call for Papers

Deadline for paper submission: 15 June, 1999

Registration fee is 100 US $ for regular participants (10.000 SIT for participants from Slovenia) and 50 US $ for students (3.500 SIT for Slovenian students). The fee covers conference materials and refreshments during coffee-breaks.

More information

For more information visit
`http://ai.ijs.si/is/indexa.html` or contact `milica.remetic@ijs.si`.

The multi-conference consists of the following conferences:

Information Society

12–14 October, 1999
Chairs: dr. Cene Bavec, prof. dr. Matjaž Gams
Contact person: prof. dr. Matjaž Gams
Phone: (+386 61) 1773 644
E-mail: `matjaz.gams@ijs.si`
Jozef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia, Europe

Data Mining and Warehouses

Chair: dr. Dunja Mladenič, Marko Grobelnik
Contact person: Marko Grobelnik
Phone: (+386 61) 1773 272
E-mail: marko.grobelnik@ijs.si
Address: Jozef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia, Europe

Manufacturing Systems and Technologies

Chair: prof. dr. Jadran Lenarčič
Contact person: prof. dr. Jadran Lenarčič
Phone: (+386 61) 1773 378
E-mail: jadran.lenarcic@ijs.si
Address: Jozef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia, Europe

Education in Information Society

Chair: prof. dr. Vladislav Rajkovič
Contact person: Mojca Florjančič
Phone: (+386 064) 22 10 61
E-mail: mojca.florjancic@fov.uni-mb.si
Address: Faculty of Organizational Sciences, Kidričeva 55a, 4000 Kranj, Slovenia, Europe

Development and Reingeneering of Informaton Systems

Chair: prof. dr. Ivan Rozman
Contact person: dr. Ivan Rozman
Phone: (386 62) 2207 410
E-mail: i.rozman@uni-mb.si
Address: FERI, Smetanova 17, 2000 Maribor, Slovenia, Europe

Biology and Cognitive Sciences

Chair: prof. dr. Igor Jerman, mag. Alexis Zrimec
Contact person: mag. Alexis Zrimec
Phone: (061) 1769 200
E-mail: alexis.zrimec@guest.arnes.si
Address: Inštitut Bion, Celovška 264, 1000 Ljubljana, Slovenia, Europe

ERK'99

Electrotechnical and Computer Science Conference
Elektrotehniška in računalniška konferenca

September 23–25, 1999

Conference Chairman
Baldomir Zajc
University of Ljubljana
Faculty of Electrical Engineering
Tržaška 25, 1001 Ljubljana, Slovenia
Tel: (061) 1768 349, Fax: (061) 1264 630
E-mail: Baldomir.Zajc@fe.uni-lj.si

Conference Vice-chairman
Jurij Tasič
University of Ljubljana
Faculty of Electrical Engineering
Tržaška 25, 1001 Ljubljana, Slovenia
Tel: (061) 1768 440, Fax: (061) 1264 630
E-mail: Jure.Tasic@fe.uni-lj.si

Program Committee Chairman
Saša Divjak
University of Ljubljana
Faculty of Comput. and Inform. Science
Tržaška 25, 1001 Ljubljana, Slovenia
Tel: (061) 1768 260, Fax: (061) 1264 647
E-mail: Sasa.Divjak@fri.uni-lj.si

Programe Committee
Tadej Bajd
Genevieve Baudoin
Gerry Cain
Saša Divjak
Janko Drnovšek
Matjaž Gams
Ferdo Gubina
Marko Jagodič
Drago Matko
Miro Milanovič
Andrej Novak
Nikola Pavešić
Franjo Pernuš
Kurt Richter
Borut Zupančič

Publications Chairman
Franc Solina
University of Ljubljana
Faculty of Comput. and Inform. Science
Tržaška 25, 1001 Ljubljana, Slovenia
Tel: (061) 1768 389, Fax: (061) 1264 647
E-mail: Franc.Solina@fri.uni-lj.si

Advisory Board
Rudi Bric
Damjan Dittrich
Karel Jezernik

Call for Papers

for the eighth **Electrotechnical and Computer Science Conference ERK'99**, which will be held on 23–25 September 1999 in Portorož, Slovenia.

The following areas will be represented at the conference:

- *electronics,*
- *telecommunications,*
- *automatic control,*
- *simulation and modeling,*
- *robotics,*
- *computer and information science,*
- *artificial intelligence,*
- *pattern recognition,*
- *biomedical engineering,*
- *power engineering,*
- *measurements,*
- *didactics.*

The conference is organized by the **IEEE Slovenia Section** together with the Slovenian Electrotechnical Society and other Slovenian professional societies:

- Slovenian Society for Automatic Control,
- Slovenian Measurement Society,
- SLOKO–CIGRE,
- Slovenian Society for Medical and Biological Engineering,
- Slovenian Society for Robotics,
- Slovenian Artificial Intelligence Society,
- Slovenian Pattern Recognition Society,
- Slovenian Society for Simulation and Modeling.

Authors who wish to present a paper at the conference should send two copies of their final camera-ready paper to mag. Andrej Trost to Faculty of Electrical Engineering, Tržaška 25, 1001 Ljubljana. The paper should be max. four pages long. More information on `http://www.ieee.si/erk99/`

Time schedule: Camera-ready paper due: *July 20, 1999*
Notification of acceptance: *End of August, 1999*

Call For Papers

8th International Conference on Computer Analysis of Images and Patterns
CAIP'99

Ljubljana, Slovenia, 1-3 September 1999

Conference Cochairs

Franc Solina, Aleš Leonardis
University of Ljubljana
Faculty of Comput. and Inform. Science
Tržaška 25,
1001 Ljubljana, Slovenia
Tel:386 61 1768 389,
Fax:386 61 1264 647,
E-mail:
franc.solina,ales.leonardis@fri.uni-lj.si

Program Committee

S. Ablameyko, Belarus
J. Arnspang, Denmark
R. Bajcsy, USA
I. Bajla, Slovakia
A. M. Bruckstein, Israel
V. Chernov, Russia
D. Chetverikov, Hungary
A. Del Bimbo, Italy
J. O. Eklundh, Sweden
V. Hlaváč, Czech Republic
J. Kittler, United Kingdom
R. Klette, New Zealand
W. Kropatsch, Austria
A. Leonardis, Slovenia
R. Mohr, France
M. Schlesinger, Ukraine
W. Skarbek, Poland
F. Solina, Slovenia
G. Sommer, Germany
L. Van Gool, Belgium
M. A. Viergever, Netherlands
S. W. Zucker, USA

Call for Papers

The CAIP conference is a traditional Central European Conference devoted to all aspects of computer vision, image analysis, pattern recognition and related fields.

The conference is sponsored by IAPR, Slovenian Pattern Recognition Society, IEEE Slovenia Section, Faculty of Computer and Information Science at the University of Ljubljana and Hermes SoftLab.

The scientific program of the conference will consist of plenary lectures by invited speakers, contributed papers presented in two parallel sessions and posters. The CAIP proceedings are published by Springer Verlag in the series Lecture Notes on Computer Science and will be distributed to the participants at the conference.

Scope of the Conference

- Image Analysis
- Computer Vision
- Pattern Recognition
- Medical Imaging
- Network Centric Vision
- Augmented Reality
- Image and Video Indexing
- Industrial Applications

Instructions to authors

Authors who wish to present a paper at the conference should send five copies of their paper to one of the two conference chairs marked CAIP'99. To enable double blind review there should be two title pages. The first with title, author's name, affiliation and address, telephone, fax and e-mail, abstract of 200 words and up to three keywords. The second title page should consist only of title, abstract and keywords. The papers excluding the title pages should not be longer than 10 pages. On a separate page the authors should answer the following three questions about their paper: (a) what is the original contribution?, (b) what is the most similar work?, (c) why is their work relevant to others?

A LaTeX template for the camera-ready version will be available on the conference home page. During the CAIP'99 review period the authors should not submitt any related paper with essentially the same content to any other conference.

Deadline for submission of papers: 15 January 1999

Registration

Information on registration will be available on the conference home page.

Venue

The conference will be held at the Faculty of Computer and Information Science at the University of Ljubljana. Ljubljana is the capital of Slovenia. The city which is a lively mixture of Mediterranean and northern influences offers all amenities within short distance. Alpine resorts, the Adriatic coast and several natural spas are close to Ljubljana.
The conference homepage is:
`http://razor.fri.uni-lj.si/CAIP99`

Machine Learning List

The Machine Learning List is moderated. Contributions should be relevant to the scientific study of machine learning. Mail contributions to `ml@ics.uci.edu`. Mail requests to be added or deleted to `ml-request@ics.uci.edu`. Back issues may be FTP'd from `ics.uci.edu` in `pub/ml-list/V<X>/<N>` or N.Z where X and N are the volume and number of the issue; ID: anonymous PASSWORD: <your mail address> URL-`http://www.ics.uci.edu/AI/ML/Machine-Learning.html`

Call for Papers

Special Issue on Group Support Systems

Informatica - An International Journal of Computing and Informatics

A special issue of the Informatica Journal will focus on the different aspects of Group Support Systems. Original, unpublished contributions and invited articles will be considered for the issue.

As organizations are being directed to do more with less, personnel productivity issues are becoming more important. No single person has the knowledge, time, or experience to solve today's business problems, so organizations have adopted a team approach. One of the challenges to this team approach is to ensure the proper make-up of the teams. Getting the maximum mix of personnel usually means picking team members from across the organization. This can pose problems when the organization is national, or global in geographical span. Technology to support these teams has been in use for some time now, and research is beginning to emerge showing how successful this merging of technology and project teams has been

To be successful, research in this area should address issues relating to the technology. It should also focus on the effectiveness to the group, and the degree of success in solving problems. Topics of interest to this special issue will include, but not be limited to, the following:

- Critical success factors for implementing group support systems
- Factors affecting the diffusion of group support systems
- Reliable measures for predicting successful implementation
- Cross-cultural factors affecting use
- Management issues with applying group support systems
- Successful implementations
- Uses for group support systems in industry
- Perceived usefulness among project teams
- Ease of use issues
- Ease of implementation issues
- Required levels of technology support
- Barriers to implementation and adoption
- Any other interesting topic that is relevant to group support systems

Prospective authors should follow the regular guidelines of the Journal and submit their work electronically. You should e-mail your rtf or PDF file to one of the Guest Editors by the following due dates:
Full Manuscript Due May 15, 1999
Notification of Acceptance September 15, 1999
Final revisions of accepted papers December 1, 1999

Gary Klein or Morgan Shepherd, Guest Editors
College of Business and Administration
The University of Colorado at Colorado Springs
1420 Austin Bluffs Parkway
P.O. Box 7150
Colorado Springs, CO 80933-7150
Gklein@mail.uccs.edu or mshepher@mail.uccs.edu
Marcin Paprzycki
Department of Computer Science and Statistics
University of Southern Mississippi
Hattiesburg, MS 39406-5106, USA
phone: 601-266-6639 or: 601-266-4949
FAX: 601-266-6452

Call for Papers

Special Issue on Design Issues of Gigabit Networking

Informatica - An International Journal of Computing and Informatics

A special issue of the Informatica Journal will focus on the different aspects of the design of Gigabit networking. Original, unpublished contributions and invited articles will be considered for the issue. With the advent of the World Wide Web, the Internet has seen enormous growth from its roots as a network of modest proportions, mostly used by the research and academic community, to a large public data network. Several thousands of corporate users and several millions of dial-in residential users have gone online in the last few years, making the Internet a true public data network. This accelerated growth in subscription has led to a surge of data in the Internet backbone. In order to keep up with this demand in service and bandwidth, all Internet service providers have scaled their networks many times, in both size and bandwidth. The forecast for this continuing growth is even more astounding for the coming years. With fast emerging technologies, such as transfer of multimedia and electronic commerce, the need to scale the network beyond present capabilities is paramount. For this the Internet has to scale in several dimensions, including but not limited to bandwidth, routing, quality of service (QoS), and customer service provisioning. To be able to succeed, research has to investigate issues of the backbone network (SONET/SDH). It should also focus on the reliability of the network and the transparency of any self-healing to the user. Topics of interest to this special issue will include, but not limited to, the following:

- Design infrastructure of Gigabit networking (physical, data link and network issues)
- SONET/SDH architectures
- Protocol design for multimedia
- Fault-tolerant of backbone networks
- Routing and congestion protocols in Gigabit networking
- Switching issues in Gigabit networking
- QoS issues in using ATM in Gigabit networking
- Modeling and simulation
- Performance evaluation in Gigabit networking
- Management issues in Gigabit networking
- Any other interesting topic that is relevant to the backbone infrastructure design

Prospective authors should follow the regular guidelines of the Journal and submit their work electronically. You should e-mail your PDF or postscript file (preferably produced by dvips and viewable by ghostview) to one of the Guest Editors listed below:

Full Manuscript Due January 31, 1999
Notification of Acceptance July 31, 1999

Mohsen Guizani, Guest Editor
Electrical and Computer Engineering Architect
University of Missouri-Columbia/Kansas City
5605 Troost Avenue
Kansas City, MO 64110-2823.
E-mail: guizanim@umkc.edu
Kenneth Henriksen, Guest Editor
Chief Technology Integration Architect-Sprint
M/S: KSOPKB0803
9300 Metcalf Avenue
Oveland Park, KS 66212, USA.
E-mail: henriksen@sprintcorp.com
http://orca.st.usm.edu/informatica

Call for Papers

Special Issue on Advances in Simulation and Control

Informatica - An International Journal of Computing and Informatics

A special issue of Informatica on the topic of "Advances in Simulation and Control" is planned for publication as one of four issues in 2000 year. As computing power has been increased, most of the real world systems can be efficiently simulated and controlled in real time with the incorporation of advanced simulation and control techniques. The issue will be intended to serve as a medium for exchanging the latest research trends in the areas of "Simulation and Control."

Papers describing the state-of-the-art research on various simulation and control topics including, but not limited to, the following areas are solicited:

- Advances in Simulation and Control Methodology,
- Fuzzy and Neural Network Techniques in Simulation and Control,
- Real-time and Distributed Simulation and Control,
- Advanced Man-Machine Interfaces for Efficient Simulation and Control,
- Simulation and Control Applications in Complex Physical Systems such as Electricity Generating Power Plants and Power Systems.

Prospective authors should follow the regular guidelines of the Journal and submit their work electronically (by e-mail) to one of guest editors by the following due dates:

Important Schedules: Full papers are due May 7, 1999 in any format of PDF, PS, MS Word or WordPerfect, with author notification set for November 20, 1999. Final revisions of accepted papers are due February 26, 2000, only in "LaTex" format. Authors interested in submitting a paper for the issue should contact one of the guest editors listed below for submission details.

Robert M. Edwards, Guest Editor
Nuclear Engineering Department
231 Sackett Building
Pennsylvania State University
University Park, PA, 16802, USA.
Phone: +1 (814) 865-0037
FAX: +1 (814) 865-8499
Email: rmenuc@engr.psu.edu
Kwang Y. Lee, Guest Editor
Department of Electrical Engineering
The Pennsylvania State University
University Park, PA 16802, USA.
Phone: +1 (814) 865-2621
Fax: +1 (814) 865-7065
E-mail: kylece@engr.psu.edu
Se Woo Cheon, Guest Editor
Korea Atomic Energy Research Institute
Dukjin 150, Yuseong, Taejon 305-353, KOREA
Phone: +82-42-868-2261
Fax: +82-42-868-8357
E-mail: swcheon@nanum.kaeri.re.kr

THE MINISTRY OF SCIENCE AND TECHNOLOGY OF THE REPUBLIC OF SLOVENIA

Address: Slovenska 50, 1000 Ljubljana, Tel.: +386 61 1311 107, Fax: +386 61 1324 140.
WWW:http://www.mzt.si
Minister: Lojze Marinček, Ph.D.

The Ministry also includes:

The Standards and Metrology Institute of the Republic of Slovenia
Address: Kotnikova 6, 61000 Ljubljana, Tel.: +386 61 1312 322, Fax: +386 61 314 882.

Slovenian Intellectual Property Office
Address: Kotnikova 6, 61000 Ljubljana, Tel.: +386 61 1312 322, Fax: +386 61 318 983.

Office of the Slovenian National Commission for UNESCO
Address: Slovenska 50, 1000 Ljubljana, Tel.: +386 61 1311 107, Fax: +386 61 302 951.

Scientific, Research and Development Potential:

The Ministry of Science and Technology is responsible for the R&D policy in Slovenia, and for controlling the government R&D budget in compliance with the National Research Program and Law on Research Activities in Slovenia. The Ministry finances or co-finance research projects through public bidding, while it directly finance some fixed cost of the national research institutes.

According to the statistics, based on OECD (Frascati) standards, national expenditures on R&D raised from 1,6 % of GDP in 1994 to 1,71 % in 1995. Table 2 shows an income of R&D organisation in million USD.

Objectives of R&D policy in Slovenia:

- maintaining the high level and quality of scientific technological research activities;
- stimulation and support to collaboration between research organisations and business, public, and other sectors;

Total investments in R&D (% of GDP)	1,71
Number of R&D Organisations	297
Total number of employees in R&D	12.416
Number of researchers	6.094
Number of Ph.D.	2.155
Number of M.Sc.	1.527

Table 1: Some R&D indicators for 1995

	Ph.D.			M.Sc.		
	1993	1994	1995	1993	1994	1995
Bus. Ent.	51	93	102	196	327	330
Gov. Inst.	482	574	568	395	471	463
Priv. np Org.	10	14	24	12	25	23
High. Edu.	1022	1307	1461	426	772	711
TOTAL	1565	1988	2155	1029	1595	1527

Table 2: Number of employees with Ph.D. and M.Sc.

- stimulating and supporting of scientific and research disciplines that are relevant to Slovenian national authenticity;
- co-financing and tax exemption to enterprises engaged in technical development and other applied research projects;
- support to human resources development with emphasis on young researchers; involvement in international research and development projects;
- transfer of knowledge, technology and research achievements into all spheres of Slovenian society.

Table source: Slovene Statistical Office.

	Basic Research		Applied Research		Exp. Devel.		Total	
	1994	1995	1994	1995	1994	1995	1994	1995
Business Enterprises	6,6	9,7	48,8	62,4	45,8	49,6	101,3	121,7
Government Institutes	22,4	18,6	13,7	14,3	9.9	6,7	46,1	39,6
Private non-profit Organisations	0,3	0,7	0,9	0,8	0,2	0,2	1,4	1,7
Higher Education	17,4	24,4	13,7	17,4	8,0	5,7	39,1	47,5
TOTAL	46,9	53,4	77,1	94,9	63.9	62,2	187,9	210,5

Table 3: Incomes of R&D organisations by sectors in 1995 (in million USD)

INFORMATICA

AN INTERNATIONAL JOURNAL OF COMPUTING AND INFORMATICS

INVITATION, COOPERATION

Submissions and Refereeing

Please submit three copies of the manuscript with good copies of the figures and photographs to one of the editors from the Editorial Board or to the Contact Person. At least two referees outside the author's country will examine it, and they are invited to make as many remarks as possible directly on the manuscript, from typing errors to global philosophical disagreements. The chosen editor will send the author copies with remarks. If the paper is accepted, the editor will also send copies to the Contact Person. The Executive Board will inform the author that the paper has been accepted, in which case it will be published within one year of receipt of e-mails with the text in Informatica LaTeX format and figures in `.eps` format. The original figures can also be sent on separate sheets. Style and examples of papers can be obtained by e-mail from the Contact Person or from FTP or WWW (see the last page of Informatica).

Opinions, news, calls for conferences, calls for papers, etc. should be sent directly to the Contact Person.

QUESTIONNAIRE

☐ Send Informatica free of charge

☐ Yes, we subscribe

Please, complete the order form and send it to Dr. Rudi Murn, Informatica, Institut Jožef Stefan, Jamova 39, 61111 Ljubljana, Slovenia.

Since 1977, Informatica has been a major Slovenian scientific journal of computing and informatics, including telecommunications, automation and other related areas. In its 16th year (more than seven years ago) it became truly international, although it still remains connected to Central Europe. The basic aim of Informatica is to impose intellectual values (science, engineering) in a distributed organisation.

Informatica is a journal primarily covering the European computer science and informatics community - scientific and educational as well as technical, commercial and industrial. Its basic aim is to enhance communications between different European structures on the basis of equal rights and international refereeing. It publishes scientific papers accepted by at least two referees outside the author's country. In addition, it contains information about conferences, opinions, critical examinations of existing publications and news. Finally, major practical achievements and innovations in the computer and information industry are presented through commercial publications as well as through independent evaluations.

Editing and refereeing are distributed. Each editor can conduct the refereeing process by appointing two new referees or referees from the Board of Referees or Editorial Board. Referees should not be from the author's country. If new referees are appointed, their names will appear in the Refereeing Board.

Informatica is free of charge for major scientific, educational and governmental institutions. Others should subscribe (see the last page of Informatica).

ORDER FORM – INFORMATICA

Name: .....

Title and Profession (optional): .....

.....

Home Address and Telephone (optional): .....

.....

Office Address and Telephone (optional): .....

.....

E-mail Address (optional): .....

Signature and Date: .....

JOŽEF STEFAN INSTITUTE

Jožef Stefan (1835-1893) was one of the most prominent physicists of the 19th century. Born to Slovene parents, he obtained his Ph.D. at Vienna University, where he was later Director of the Physics Institute, Vice-President of the Vienna Academy of Sciences and a member of several scientific institutions in Europe. Stefan explored many areas in hydrodynamics, optics, acoustics, electricity, magnetism and the kinetic theory of gases. Among other things, he originated the law that the total radiation from a black body is proportional to the 4th power of its absolute temperature, known as the Stefan–Boltzmann law.

The Jožef Stefan Institute (JSI) is the leading independent scientific research institution in Slovenia, covering a broad spectrum of fundamental and applied research in the fields of physics, chemistry and biochemistry, electronics and information science, nuclear science technology, energy research and environmental science.

The Jožef Stefan Institute (JSI) is a research organisation for pure and applied research in the natural sciences and technology. Both are closely interconnected in research departments composed of different task teams. Emphasis in basic research is given to the development and education of young scientists, while applied research and development serve for the transfer of advanced knowledge, contributing to the development of the national economy and society in general.

At present the Institute, with a total of about 700 staff, has 500 researchers, about 250 of whom are postgraduates, over 200 of whom have doctorates (Ph.D.), and around 150 of whom have permanent professorships or temporary teaching assignments at the Universities.

In view of its activities and status, the JSI plays the role of a national institute, complementing the role of the universities and bridging the gap between basic science and applications.

Research at the JSI includes the following major fields: physics; chemistry; electronics, informatics and computer sciences; biochemistry; ecology; reactor technology; applied mathematics. Most of the activities are more or less closely connected to information sciences, in particular computer sciences, artificial intelligence, language and speech technologies, computer-aided design, computer architectures, biocybernetics and robotics, computer automation and control, professional electronics, digital communications and networks, and applied mathematics.

The Institute is located in Ljubljana, the capital of the independent state of **Slove**nia (or S♡nia). The capital today is considered a crossroad between East, West and Mediterranean Europe, offering excellent productive capabilities and solid business opportunities, with strong international connections. Ljubljana is connected to important centers such as Prague, Budapest, Vienna, Zagreb, Milan, Rome, Monaco, Nice, Bern and Munich, all within a radius of 600 km.

In the last year on the site of the Jožef Stefan Institute, the Technology park "Ljubljana" has been proposed as part of the national strategy for technological development to foster synergies between research and industry, to promote joint ventures between university bodies, research institutes and innovative industry, to act as an incubator for high-tech initiatives and to accelerate the development cycle of innovative products.

At the present time, part of the Institute is being reorganized into several high-tech units supported by and connected within the Technology park at the Jožef Stefan Institute, established as the beginning of a regional Technology park "Ljubljana". The project is being developed at a particularly historical moment, characterized by the process of state reorganisation, privatisation and private initiative. The national Technology Park will take the form of a shareholding company and will host an independent venture-capital institution.

The promoters and operational entities of the project are the Republic of Slovenia, Ministry of Science and Technology and the Jožef Stefan Institute. The framework of the operation also includes the University of Ljubljana, the National Institute of Chemistry, the Institute for Electronics and Vacuum Technology and the Institute for Materials and Construction Research among others. In addition, the project is supported by the Ministry of Economic Relations and Development, the National Chamber of Economy and the City of Ljubljana.

Jožef Stefan Institute
Jamova 39, 61000 Ljubljana, Slovenia
Tel.:+386 61 1773 900, Fax.:+386 61 219 385
Tlx.:31 296 JOSTIN SI
WWW: http://www.ijs.si
E-mail: matjaz.gams@ijs.si
Contact person for the Park: Iztok Lesjak, M.Sc.
Public relations: Natalija Polenec

Informatica WWW:

http://ai.ijs.si/Mezi/informatica.htm
http://orca.st.usm.edu/informatica/

Referees:

Witold Abramowicz, David Abramson, Kenneth Aizawa, Suad Alagić, Alan Aliu, Richard Amoroso, John Anderson, Hans-Jurgen Appelrath, Grzegorz Bartoszewicz, Catriel Beeri, Daniel Beech, Fevzi Belli, Istvan Berkeley, Azer Bestavros, Balaji Bharadwaj, Jacek Blazewicz, Laszlo Boeszoermenyi, Damjan Bojadžijev, Jeff Bone, Ivan Bratko, Jerzy Brzezinski, Marian Bubak, Leslie Burkholder, Frada Burstein, Wojciech Buszkowski, Netiva Caftori, Jason Ceddia, Ryszard Choras, Wojciech Cellary, Wojciech Chybowski, Andrzej Ciepielewski, Vic Ciesielski, David Cliff, Travis Craig, Noel Craske, Matthew Crocker, Tadeusz Czachorski, Milan Češka, Honghua Dai, Andrej Dobnikar, Sait Dogru, Georg Dorfner, Ludoslaw Drelichowski, Matija Drobnič, Maciej Drozdowski, Marek Druzdzel, Jozo Dujmović, Pavol Ďuriš, Hesham El-Rewini, Pierre Flener, Wojciech Fliegner, Vladimir A. Fomichov, Terrence Forgarty, Hans Fraaije, Hugo de Garis, Eugeniusz Gatnar, James Geller, Michael Georgiopolus, Jan Goliński, Janusz Gorski, Georg Gottlob, David Green, Herbert Groiss, Inman Harvey, Elke Hochmueller, Rod Howell, Tomáš Hruška, Alexey Ippa, Ryszard Jakubowski, Piotr Jedrzejowicz, Eric Johnson, Polina Jordanova, Djani Juričič, Sabhash Kak, Li-Shan Kang, Roland Kaschek, Jan Kniat, Stavros Kokkotos, Kevin Korb, Gilad Koren, Henryk Krawczyk, Ben Kroese, Zbyszko Krolikowski, Benjamin Kuipers, Matjaž Kukar, Aarre Laakso, Phil Laplante, Bud Lawson, Ulrike Leopold-Wildburger, Joseph Y-T. Leung, Alexander Linkevich, Raymond Lister, Doug Locke, Peter Lockeman, Matija Lokar, Jason Lowder, Andrzej Małachowski, Bernardo Magnini, Peter Marcer, Andrzej Marciniak, Witold Marciszewski, Vladimir Marik, Jacek Martinek, Tomasz Maruszewski, Florian Matthes, Timothy Menzies, Dieter Merkl, Zbigniew Michalewicz, Roland Mittermeir, Madhav Moganti, Tadeusz Morzy, Daniel Mossé, John Mueller, Hari Narayanan, Elzbieta Niedzielska, Marian Niedq'zwiedziński, Jaroslav Nieplocha, Jerzy Nogieć, Stefano Nolfi, Franc Novak, Antoni Nowakowski, Adam Nowicki, Tadeusz Nowicki, Hubert Österle, Wojciech Olejniczak, Jerzy Olszewski, Cherry Owen, Mieczyslaw Owoc, Tadeusz Pankowski, Mitja Peruš, Warren Persons, Stephen Pike, Niki Pissinou, Ullin Place, Gustav Pomberger, James Pomykalski, Gary Preckshot, Dejan Rakovič, Cveta Razdevšek Pučko, Ke Qiu, Michael Quinn, Gerald Quirchmayer, Luc de Raedt, Ewaryst Rafajlowicz, Sita Ramakrishnan, Wolf Rauch, Peter Rechenberg, Felix Redmill, David Robertson, Marko Robnik, Ingrid Russel, A.S.M. Sajeev, Bo Sanden, Vivek Sarin, Iztok Savnik, Walter Schempp, Wolfgang Schreiner, Guenter Schmidt, Heinz Schmidt, Denis Sever, William Spears, Hartmut Stadtler, Olivero Stock, Janusz Stokłosa, Przemysław Stpiczyński, Andrej Stritar, Maciej Stroinski, Tomasz Szmuc, Zdzislaw Szyjewski, Jure Šilc, Metod Škarja, Jiří Šlechta, Zahir Tari, Jurij Tasič, Piotr Teczynski, Stephanie Teufel, Ken Tindell, A Min Tjoa, Wieslaw Traczyk, Roman Trobec, Marek Tudruj, Andrej Ule, Amjad Umar, Andrzej Urbanski, Marko Uršič, Tadeusz Usowicz, Elisabeth Valentine, Kanonkluk Vanapipat, Alexander P. Vazhenin, Zygmunt Vetulani, Olivier de Vel, John Weckert, Gerhard Widmer, Stefan Wrobel, Stanislaw Wrycza, Janusz Zalewski, Damir Zazula, Yanchun Zhang, Robert Zorc, Anton P. Železnikar

EDITORIAL BOARDS, PUBLISHING COUNCIL

Informatica is a journal primarily covering the European computer science and informatics community; scientific and educational as well as technical, commercial and industrial. Its basic aim is to enhance communications between different European structures on the basis of equal rights and international refereeing. It publishes scientific papers accepted by at least two referees outside the author's country. In addition, it contains information about conferences, opinions, critical examinations of existing publications and news. Finally, major practical achievements and innovations in the computer and information industry are presented through commercial publications as well as through independent evaluations.

Editing and refereeing are distributed. Each editor from the Editorial Board can conduct the refereeing process by appointing two new referees or referees from the Board of Referees or Editorial Board. Referees should not be from the author's country. If new referees are appointed, their names will appear in the list of referees. Each paper bears the name of the editor who appointed the referees. Each editor can propose new members for the Editorial Board or referees. Editors and referees inactive for a longer period can be automatically replaced. Changes in the Editorial Board are confirmed by the Executive Editors.

The coordination necessary is made through the Executive Editors who examine the reviews, sort the accepted articles and maintain appropriate international distribution. The Executive Board is appointed by the Society Informatika. Informatica is partially supported by the Slovenian Ministry of Science and Technology.

Each author is guaranteed to receive the reviews of his article. When accepted, publication in Informatica is guaranteed in less than one year after the Executive Editors receive the corrected version of the article.

Volume 23 Number 1 April 1999 ISSN 0350-5596

Informatica

An International Journal of Computing and Informatics