“Must-Work": A Scalable Model for Parallel Recursive Problems on P2P Networks

Mourad Amad, Djamil Aïssani
Laboratory LAMOS, University of Bejaia, Algeria
E-mail: amad.mourad@gmail.com, lamos_bejaia@hotmail.com

Toufik Bellal and Hameza Amrioui University of Bretagne Occidentale (Brest), France
E-mail: toufik.Bellal@univ-brest.fr, hameza.amrioui@univ-brest.fr

Keywords: P2P, on demand dynamic hierarchical master worker, recursive problems, parallel computing

Received: July 9, 2010

Most of real world problems are cpu-time intensive; their solutions take too much time to compute using a single machine. The grid and cloud computing offer a potential solution to this problem. However, such solutions are in general expensive. An alternative solution uses P2P networks, a set of machines in the Internet which collaborate to perform the same task. Branch-and-Bound is a model of such solution, but most of parallel applications developed on P2P networks are based on the Master Worker paradigm, particularly for divide and conquer problems (D&C), where the Master divides the tasks and sends them to Workers, while the Workers execute received tasks as in the client server model. This solution suffers from the scalability problem, and, as a consequence, hierarchical Master Worker model was introduced. Scalability is improved, but it still remains a critical issue. In this paper, we propose a new generic and scalable model for parallelizing the resolution of the recursive problem (a type of divide and conquer problems) on an existing P2P architecture. As opposed to the existing hierarchical Master-Worker models, in our solution, each network node is both a Master and Worker called "Must-Work". The proposed solution uses a dynamic tree for tasks distribution; it is constructed according to a node requestor. We have evaluated our solution using The Quicksort method under the MPI platform. The results are globally satisfactory in term of time execution compared to the sequential solution.

Povzetek: Članek opisuje način reševanja rekurzivnih problemov v omrežjih vsak-z-vsakim.

1 Introduction

At the present time, the mathematical resolution of the major optimization problems (*eg. vehicle routing problem, travelling salesman, minimum spanning tree, eight queens puzzle, Knapsack, Cutting stock, ...*) remains extremely complex and/or expensive in terms of machine time. It is however possible in the majority of the cases (*some problems are excluded, eg. TSP*) to distribute calculation on several machines, each one treating one part of the problem, under the aegis of a main authority. The multiprocessor machines or the material parallelization of machines single processors constitutes the first application of this concept. This involves nevertheless a high cost of sharing the number of processors to be acquired. An alternative solution is to distribute calculation on a network, but the put question is how to distribute this calculation with efficiency, optimization and fault tolerance?

Divide and conquer (D&C) is an important design paradigm based on multi-branched recursion. It works by recursively breaking down a problem into two or more sub-problems of the same type, until they become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem [8]. In fact, recursive problems (*our case study*) can be resolved by D&C type algorithms.

The recursivity is a very interesting field of data processing; it makes it possible to solve certain problems in a very fast way and very simple. The recursive problems require many means and resources, because enormous data must be stored in the stack of machine, which causes an overflow of this stack in case of one processor. Whereas, on a network (*eg. P2P*), this problem is not posed. As examples of the famous recursive problems, we can quote: the problem of the eight reins, Hanoi turn (*see figure* 2) and the Quicksort sorting (*our case study*).

Recursive problem belongs to the divide on conquer problem type, where Master-Worker models have been proposed for their solutions. In the conventional Master-Worker paradigm, a single supervisor process manages and controls a sets of processes. Distribution of tasks is performed in two phases: **1)** the distribution from supervisor to Master process, **2)** and that from Master process to Worker processes. The computed results are performed on the reverse way. The main problem of this solution is the central supervisor, and the overload of the Master.

The hierarchical Master Worker type solutions are then introduced to ameliorate the former one. In the hierarchical Master-Worker models, Masters are required to partition the problem-space (*preparing work for the Workers*), schedule work, balance the load of the Workers to maintain efficiency [11], and correlating their output into a global result. Workers simply perform given operations. This paradigm sustained good performance [3]. However, scalability is always an open issue [12].

In this paper, we consider the parallel resolution of the recursive problems (*divide and conquer type*). We propose a scalable and generic hierarchical Master Worker model based on a distributed tree diffusion which is constructed on demand by any Master node in any P2P network. It is characterized by high dynamicity (*node can join and leave the network at any time*), scalability (*hight number of node is supported*) and genericity (*for any type of P2P network and any recursive problem*).

The remainder of this paper is organized as follows: In the second section, we present a background and related works on parallel computing techniques, especially for divide and conquer problem type, with more importance to the hierarchical Master Worker models. In the third section, we give our contribution for parallelizing the recursive problem solutions on any existing P2P architecture. We give a performance evaluation of the proposed model for Quicksort method in section 4. Finally, we conclude and give some perspectives.

2 Background and Related Works

The resolution of the numerical problems which are expensive in terms of computing is a challenge. To solve a given problem more quickly, a natural idea consists of making simultaneously several agents cooperating for its resolution, which will thus work in parallel. Parallel calculation is a technique in which several actions are carried out simultaneously, so that the time of resolution is reduced. In addition to the material components intended for parallel calculation, a support by software components is also necessary, in order to coordinate the simultaneous execution of several lines of computer program code. Such dependence is necessary, because of the existing interdependencies between the various program codes [2].

Grid computing has become an alternative to traditional supercomputing environments for developing parallel applications in recent years [9]. Master-Worker paradigm is a common model to evaluate a pool of tasks, it is used by many scientific and engineering applications like tree search algorithms, genetic algorithms, training of neural networks, stochastic optimization, parameter analysis for engineering design, Monte Carlo simulation [10].

Master Worker paradigm is an adequate solution for divide and conquer problem such as our case study (*recursive problems*). It consists of two entities: a Master and multiple Workers. The Master decomposes the problem into small tasks and distributes these tasks among a farm of Worker processes, as well as for gathering the partial results in order to produce the final result of the computation. Worker processes execute in a very simple cycle: receive a message from the Master with the next task, process the task, and send back the result to the Master. This paradigm is well adapted to the first generation of P2P networks (*eg. Napster*[1]), because Napster is composed of one server which store a resource's index (*Master*). All other peers (*Workers*) are connected to the server in order to publish/search a resource (*execute tasks*). However, the most proposed Master-Worker solutions suffer from the scalability problem (*in terms of Master bottleneck but also security: the master is busy all time*) [11], more generally they are specific for only some problems and using also specific underlying architectures.

Given this, hierarchical Master-Worker model has been proposed; it uses submasters to decrease the workload of the Master, but a problem still exists: if the number of Workers or submasters grows, the submasters also will be bottleneck because many communications appear between Workers and their submasters. Hierarchical Master Worker using a shared memory space for work managing at the submasters was introduced by GhasemiGol et all (*Linda model*) [11]. In this model, Workers execute a task and put the results in a shared memory space on the submaster. Effectively, the solution reduces communication cost, but accessing shared memory is complicated and it is not practical in large scale network. Some Workers do not work voluntary, such as free rider on file sharing P2P applications which is opposed to our contribution, where nodes are called Must-Work.

In [14], the authors propose GVGE, a shared and interactive virtual collaborative geographic environment for solving geographic problems. It is composed of three layers: resource layer, service layer and application layer. GVGE is a platform that can manage grid applications. In [13], the authors propose a java environment for developing parallel programs limited to small clusters. In [15], ParCop is proposed using Master Worker model where each node manages two types of links: permanent and temporary pathways. The former makes connections with its neighbours, and the later makes connections between the Master and Worker during computational. Pathways can be more than one hop and then message transmissions consume more CPU. In [16], the authors propose how to build new types of groups called “similarity groups" into the JNGI project [17], in order to increase the relevance of task dispatching and therefore to increase the performance of JNGI. In [19], desktop grids inspired from biological systems, a large computational tasks are broken down into sufficiently small subtasks. Each subtask is encapsulated into a mobile agent. The management of the mobile agents is not shown in the paper. A similar work has been proposed in [20]. In their paper, the authors propose a middleware for parallel-based computations across a P2P network. It is different

[1]http://www.napster.com

from our work on the tree diffusion construction process, but also the fault tolerance consideration. In [22], the authors propose a nice construction recursive model that can complete our work by integrating it on each node.

The second generation of P2P network (*eg. Gnutella*) and the third generation (*eg. Chord*) can be a good candidates for hierarchical Master-Worker models; we just need to construct a tree with efficiency and optimization.

In this paper, we propose a generic and scalable model for parallel computing which can be implemented on any existing underlying P2P architectures (*structured and unstructured*), because it is based on the construction of a tree in a connex graph (*by exchange messages between neighboring nodes*) from any node without need of global knowledge of the underlying P2P architecture. In graph theory [23], the construction of a tree from source node to a set of receivers is already feasible if the graph is connected. The overlay P2P network is a connected graph.

P2P systems know an explosive growth in the few last years [1]; they progressively replace the Client/Server architecture. P2P systems are a good solutions for problems which need higher scalability, they resist to denial of service attacks, and held the top of the paving stone on Internet (*application layer*). Unlike the centralized systems such as Client/Server, in P2P systems, it is the hosts who provide the resources which will be available on the network [21]. These resources are those of the computers, the users and the connection they have. Parallel computing is an interesting example of P2P applications; it is also one of the most important solutions for the optimization problems. Many P2P architectures have been proposed for file sharing applications (*eg. Chord [6], CAN [5] and Tapestry [7]*). Figure 1 is an example of P2P architecture, we use it in performance evaluation section.

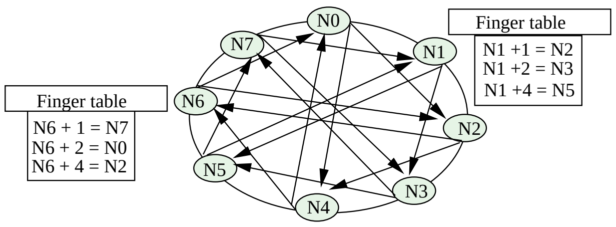


Figure 1: Example of P2P network (*Chord architecture*)

Our contribution aims to improve the scalability and contributes significantly to the genericity of the hierarchical Master Worker models by its important characteristics, it can be easily implemented on any existing P2P architecture, and then benefits from their advantages (*scalability, self organization, fault tolerance, ...*). The standard operational interpretation of a recursive program forms a tree of recursive calls spreading out from the node of the initial invocation [21]. Our solution is fundamentally based on dynamic task distribution tree constructed on demand under an existing P2P overlay network. The next section describes and analyzes the proposed solution.

3 Proposed Solution: "Must-Work"

In this paper, we propose a scalable and hierarchical Master Worker model based on a dynamic on demand tree construction, where each node is sometimes Master for a given tree, sometimes Worker for another tree, it is called Must-Work. When a node receives calculus request (*task*), it can't refuses it. If the task is complex, it plays the role of Master (*divides and conquers*). Otherwise, it is Worker (*executes task*). In some specific situations, a given node can be simultaneously Master for some Workers and Worker for some Masters. As illustrated on figure 2, the node N3 is a Master in `Hanoi(3,A,B,C)` problem resolution when N0 is a source, and it is a Worker in `Hanoi(3,A,B,C)` problem resolution when N2 is a source.

3.1 Functional principal

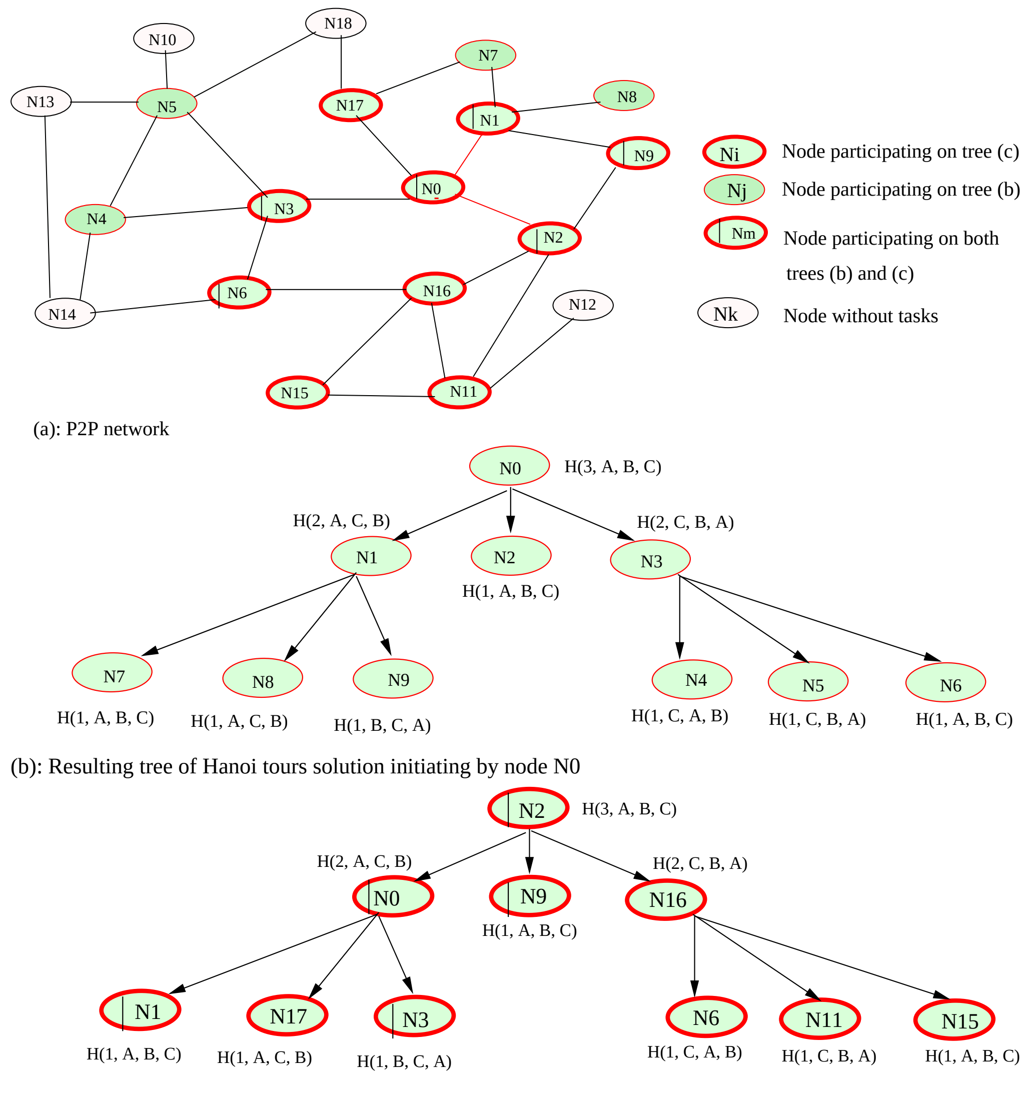
The parallelization of the recursive problems consists to break up the problem into independent calculation portions (*tasks*), and to diffuse each one on a dynamic on demand constructed tree branch from P2P underlying architecture, as long as the portion problem is decomposable. Let us arrive at an evident problem (*where its resolution is easy and does not need complex treatment or dividing*). The considered node makes calculation and returns result to the immediate requestor, and so on, until the machine initiator of calculation receives all the required calculation portions. With other words, the resolution of a recursive problem on existing P2P architecture can be transformed to a dynamic on demand diffusion tree construction, then to diffuse calculation and recovering it thereafter. The depth of the constructed diffusion tree depends on the recursive problems to solve. Figure 2 shows an example of the calculus distribution (*diffusion*) tree for the Hanoi tours problem. The network is composed of 19 nodes, the degree of the Hanoi problem is 3. Each Master divides the calculus on three parts, and sends them to their successors in the constructed tree, and so on.

Figure 2 illustrates an example for resolving simultaneously two Hanoi problems. The node N0 calculates `Hanoi (3,A,B,C)` and the node N2 calculates also `Hanoi(3,A,B,C)`, each one constructs it own tree on the same network. Node N3 participates on both resulting trees, it is a Master on the first tree (*figure 2.b*) and Worker on the second tree (*figure 2.c*).

For illustration purpose, let consider the following notations:

3.2 On demand diffusion tasks tree construction

At the time of launching calculation decision, the initiator node N_i puts its state to *initiator*, and plays the role of Master, it builds a set of receiving children (*immediate successors at one hop in the dynamic on demand constructing tree*) by executing the function `Child(i)`. Before



(a): P2P network

(b): Resulting tree of Hanoi tours solution initiating by node N0

(c): Resulting tree of Hanoi tours solution initiating by node N2

Figure 2: Diffusion tree construction for Hanoi tours problem

Notation	Designation
N_i	Node identifier.
Child(j)	Function which returns child order j of the current node (j^{th} *successor*)
Nb_nd	Number of nodes in the network.
Nb_fi	Number of children of node N_i
Ei	Set of children of node N_i (*set of successors*).
Nb_aqi	Number of acquirements for node N_i (*number of recipients which have received the request from node* N_i).
D	Number of portions (*degree*) of the recursive problem to parallelized (*Parameter of the problem to be resolved*).
state	State of node N_i (*free, occupied, initiator*) initialized at free.
Parent(i)	Predecessor of node N_i.

Table 1: Conventional notations

sending a task to any node, it divides calculation into several portions (*tasks*) according to the degree of the problem (*eg. The Hanoi turn problem shown on figure 2 is with degree 3*). The requested node sends the message: `'are you free?'` to each neighbouring. If it receives a number (NPR) of positive responses higher than the number of tasks (D) to distribute, it diffuses them towards its children (*successors*) according to algorithm 1 (*the sender node should save the set of its neighboring nodes to which it has send a task*). Otherwise, it sends the first tasks to free nodes and the remainder tasks to occupied nodes if they exist, or diffuse a second work for the same children.

As illustrated on algorithm 2, at the reception of a calculation portion from node N_j, node N_i puts its state to *occupied*, it tests if the portion of the received calculus (*task*) is not decomposable, so yes (*node is a Worker*): it makes the last calculation, and returns back the result to its parent (*transmitting of calculation result using function parent(j)*), then it changes its state to *free*. Otherwise (*node is a Master*), it breaks up this portion into a set of sub portions (*eg. k sub portions*) and diffuses them towards the set of its children (*k successors*) as illustrated on algorithm 1.

When a node N_i receives a calculus result from one child, it decreases the number of the acquirements (*Nb_aqi–*), and when it receives all the sent calculus por-

Algorithm 1 : Starting the calculus by the initiator N_i (*root*)

```
1: Begin
2: Ei = ;
3: statei = initiator;
4: Nb_fi = card({neighboring nodes}) ;
5: Ei← {neighboring nodes};
6:If ($Nb\_fi \geq D$) Then
7:     For m:=1 To D Do
8:     k← identifier of the neighboring node (Ei)
       with higher capability;
9:     Send the $k^{th}$ calculus portion to child (k);
10:    Ei←Ei - child (k);
11:    End
12:Else
13:     For k:=1 To Nb_fi Do
14:     Send the $K^{th}$ calculus portion to child (k);
15:     End
16:     For k:=Nb_fi +1 To D Do
17:     Random ($N_p$);// here, we can choose the
        better nodes in terms of calculus capabilities;
18:    Send the $K^{th}$ calculus portion to child ($N_p$);
19:     End
20:     Nb_aqi= D;
21: End
22: statei = free;
23: End.
```

Algorithm 2 :At the reception of a calculus portion by node N_i from node N_j

```
1: Begin
2: Nb_aqi = 0;
3: Ei = ∅;
4: statei = occupied;
5: perei = j;
6:     If (the calculus portion is not decomposable) Then
7:         Do the calculus and return the result to perei (requestor);
8:     statei = free;
9:     Else
10:      Execute Algorithm 1;
11:      End
12: End.
```

Algorithm 3 : At the reception of each portion calculus result

```
1: Begin
2:     Nb_aqi–;
3:     If (Nb_aqi=0) Then
4:         Do its calculus portion;
5:          If (statei= initiator) Then
6:          Return the final result
7:          Else
8:          Send the result to its parent (parent(i));
9:         End
10:         statei=free;
11:      Else
12:      Wait the other results;
13:      End
14: End.
```

tion results (*Nb_aqi=0*), it makes its calculation, and returns the final result if it is the initiator. Otherwise, it sends the result to the requestor node (*parent(i)*), and so on. This process is illustrated on algorithm 3.

3.3 Some considerations on the solution

In this section, we give some detail illustrations on the scalability, fault tolerance, load balancing of our proposed solution.

- **Scalability:** The nodes in the network are sometimes Masters, sometimes Workers; they have the same responsibility (*function*). They divide the calculus and distribute it to their successors (*neighboring nodes with one hop*), which forms progressively and dynamically a calculus diffusion tree, where only the leaf nodes (*Workers*) in the tree which does operational calculus. The inner nodes divide and conquer tasks (*Masters*). In fact, the solution does not suffer from the scalability problem, but it depends on the scalability of the underlying P2P overlay network on which is implemented (*It is more scalable on Chord architecture than on Gnutella, because Chord is more scalable than Gnutella*). On the other hand, the proposed solution is generic relatively to both underlying architecture and recursive problem to be solved.

- **Fault tolerance:** When a Master node distributes the calculus portion to its children (*successors*), it activates a predefined time-out (T), it waits until time out expiration, if it doesn't receives all calculus portion results, then it sends request to children (*Workers*) which have not returned results yet, asking them to give results. If they respond to the request and they does not finished their tasks, the requestor node increments the time-out (T). Otherwise (*the direct successor who has not given response is failed*), the requestor node redistributes their calculus portions (*supposed failed*) to other successors, and so on. The calculus is still continuous even with node failures.

  At each node, we associate a queue for storing the requested calculus (*tasks*) which have not served yet with FIFO service politic. Perhaps, the failure of the first Master (*root*) blocks the system, and the results can not be recuperated. However, in practical applications, root Master is managed by the users of application themselves, and then we assume that is a reliable node.

- **Load balancing:** In the proposed hierarchical Master-Worker solution; each node is a Master for some Workers, and a Worker for some Masters (*"Must Work"*). No node is more important than other nodes. The Master divides the complex calculus (*we assume here that each node is enough intelligent to do this* *treatment for any recursive problem*), and then diffuses the calculus portions (*tasks*) to its children (*successors at one hop*). When it receives the result portions, it makes its treatment, and then sends the result to the requestor. When a node receives a calculation request which is not decomposable, it executes it locally, and returns the result to the requestor. In the case where all the children of a given Master are on state *occupied*, and this last has not achieved the distribution of all the calculus portions, it can send them to those with high capabilities using a scheduling strategy such as in [4].

4 Performance Evaluation

In order to evaluate the proposed solution, we tested it for sorting a vector with different sizes and using the Quicksort method. We use the MPI platform (*Message Passing Interface*)[2] under linux, which is a library of C/Fortran functions based on the message communications (*the most important are: MPI_send and MPI_recv*). Simulations are down on machine carried out in a personal computer with the following characteristics: 2.16 GHz and 1GB of RAM. A specific tool was developed for simulation purposes.

Figures 3 and 4 represent respectively the sorting execution times of two vectors with dimensions 15 and 500 by the Quicksort method, using both parallel program and sequential one implemented on Chord architecture.

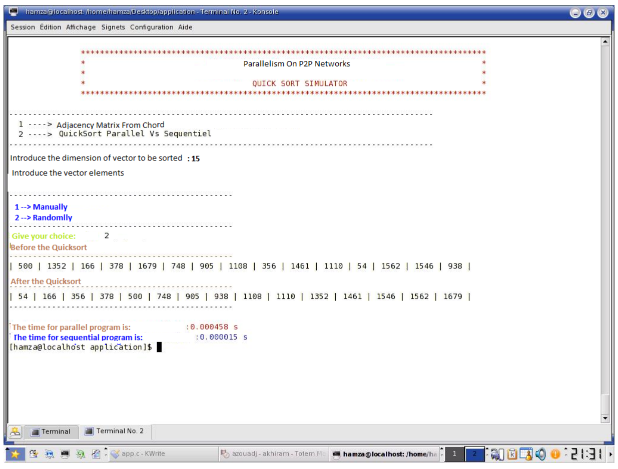

```
                 ****************************************************************
                 *                    Parallelism On P2P Networks               *
                 *                                                              *
                 *                    QUICK SORT SIMULATOR                      *
                 ****************************************************************

-----------------------------------------------------------------------------
  1 ----> Adjacency Matrix From Chord
  2 ----> QuickSort Parallel Vs Sequentiel
-----------------------------------------------------------------------------
Introduce the dimension of vector to be sorted  : 15
 Introduce the vector elements

------------------------------------------
  1 --> Manually
  2 --> Randomlly
------------------------------------------
 Give your choice:        2
Before the Quicksort
------------------------------------------
| 500 | 1352 | 166 | 378 | 1679 | 748 | 905 | 1108 | 356 | 1461 | 1110 | 54 | 1562 | 1546 | 938 |
After the Quicksort
------------------------------------------
| 54 | 166 | 356 | 378 | 500 | 748 | 905 | 938 | 1108 | 1110 | 1352 | 1461 | 1546 | 1562 | 1679 |
------------------------------------------

 The time for parallel program is:              :0.000458 s
 The time for sequential program is:              :0.000015 s
[hamza@localhost application]$ 
```

Figure 3: Sorting a vector of 15 elements using Quicksort method

For the vector of size 15 (*Figure 3*), the execution time for the sorting using the sequential solution is 0.000015s, and that using the parallel solution is 0.000458s. Sequential solution is better than parallel one. This is du to the high cost of message communication time comparatively to the execution time.

For a vector with 500 elements (*Figure 4*), the execution time for the sequential solution is 0.00326s. Whereas, the

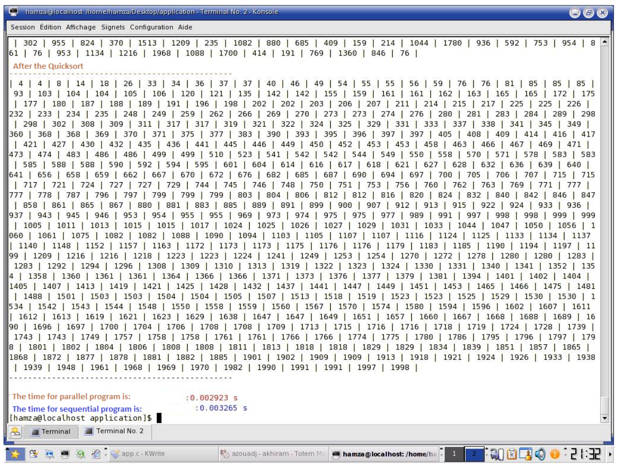

```
 | 302 | 955 | 824 | 370 | 1513 | 1209 | 235 | 1082 | 880 | 685 | 409 | 159 | 214 | 1044 | 1780 | 936 | 592 | 753 | 954 | 8
61 | 76 | 953 | 1134 | 1216 | 1968 | 1088 | 1700 | 414 | 191 | 769 | 1360 | 846 | 76 |
 After the Quicksort
------------------------------------------
| 4 | 4 | 8 | 14 | 18 | 26 | 33 | 34 | 36 | 37 | 37 | 40 | 46 | 49 | 54 | 55 | 55 | 56 | 59 | 76 | 76 | 81 | 85 | 85 | 85 |
 93 | 103 | 104 | 104 | 105 | 106 | 120 | 121 | 135 | 142 | 142 | 155 | 159 | 161 | 161 | 162 | 163 | 165 | 165 | 172 | 175
 | 177 | 180 | 187 | 188 | 189 | 191 | 196 | 198 | 202 | 202 | 203 | 206 | 207 | 211 | 214 | 215 | 217 | 225 | 225 | 226 |
232 | 233 | 234 | 235 | 248 | 249 | 259 | 262 | 266 | 269 | 270 | 273 | 273 | 274 | 276 | 280 | 281 | 283 | 284 | 289 | 298
 | 298 | 302 | 308 | 309 | 311 | 317 | 317 | 319 | 321 | 322 | 324 | 325 | 329 | 331 | 333 | 337 | 338 | 341 | 345 | 349 |
360 | 368 | 368 | 369 | 370 | 371 | 375 | 377 | 383 | 390 | 393 | 395 | 396 | 397 | 397 | 405 | 408 | 409 | 414 | 416 | 417
 | 421 | 427 | 430 | 432 | 435 | 436 | 441 | 445 | 446 | 449 | 450 | 452 | 453 | 453 | 458 | 463 | 466 | 467 | 469 | 471 |
473 | 474 | 483 | 486 | 486 | 499 | 499 | 510 | 523 | 541 | 542 | 542 | 544 | 549 | 550 | 558 | 570 | 571 | 578 | 583 | 583
 | 585 | 588 | 588 | 590 | 592 | 594 | 595 | 601 | 604 | 614 | 616 | 617 | 618 | 621 | 627 | 628 | 632 | 636 | 639 | 640 |
641 | 656 | 658 | 659 | 662 | 667 | 670 | 672 | 676 | 682 | 685 | 687 | 690 | 694 | 697 | 700 | 705 | 706 | 707 | 715 | 715
 | 717 | 721 | 724 | 727 | 727 | 729 | 744 | 745 | 746 | 748 | 750 | 751 | 753 | 756 | 760 | 762 | 763 | 769 | 771 | 777 |
777 | 778 | 787 | 796 | 797 | 799 | 799 | 799 | 803 | 804 | 806 | 812 | 812 | 816 | 820 | 824 | 832 | 840 | 842 | 846 | 847
 | 858 | 861 | 865 | 867 | 880 | 881 | 883 | 885 | 889 | 891 | 899 | 900 | 907 | 912 | 913 | 915 | 922 | 924 | 933 | 936 |
937 | 943 | 945 | 946 | 953 | 954 | 955 | 955 | 969 | 973 | 974 | 975 | 975 | 977 | 989 | 991 | 997 | 998 | 998 | 999 | 999
 | 1005 | 1011 | 1013 | 1015 | 1015 | 1017 | 1024 | 1025 | 1026 | 1027 | 1029 | 1031 | 1033 | 1044 | 1047 | 1050 | 1056 | 1
060 | 1061 | 1075 | 1082 | 1082 | 1088 | 1090 | 1094 | 1103 | 1105 | 1107 | 1107 | 1116 | 1124 | 1125 | 1133 | 1134 | 1137
| 1140 | 1148 | 1152 | 1157 | 1163 | 1172 | 1173 | 1173 | 1175 | 1176 | 1176 | 1179 | 1183 | 1185 | 1190 | 1194 | 1197 | 11
99 | 1209 | 1216 | 1216 | 1218 | 1223 | 1223 | 1224 | 1241 | 1249 | 1253 | 1254 | 1270 | 1272 | 1278 | 1280 | 1280 | 1283 |
 1283 | 1292 | 1294 | 1296 | 1308 | 1309 | 1310 | 1313 | 1319 | 1322 | 1323 | 1324 | 1330 | 1331 | 1340 | 1341 | 1352 | 135
4 | 1358 | 1360 | 1361 | 1361 | 1364 | 1366 | 1366 | 1371 | 1373 | 1376 | 1377 | 1379 | 1381 | 1394 | 1401 | 1402 | 1404 |
1405 | 1407 | 1413 | 1419 | 1421 | 1425 | 1428 | 1432 | 1437 | 1441 | 1447 | 1449 | 1451 | 1453 | 1465 | 1466 | 1475 | 1481
 | 1488 | 1501 | 1503 | 1503 | 1504 | 1504 | 1505 | 1507 | 1513 | 1518 | 1519 | 1523 | 1523 | 1525 | 1529 | 1530 | 1530 | 1
534 | 1542 | 1543 | 1544 | 1548 | 1550 | 1558 | 1559 | 1560 | 1567 | 1570 | 1574 | 1580 | 1594 | 1596 | 1602 | 1607 | 1611
| 1612 | 1613 | 1619 | 1621 | 1623 | 1629 | 1638 | 1647 | 1647 | 1649 | 1651 | 1657 | 1660 | 1667 | 1668 | 1688 | 1689 | 16
90 | 1696 | 1697 | 1700 | 1704 | 1706 | 1708 | 1708 | 1709 | 1713 | 1715 | 1716 | 1716 | 1718 | 1719 | 1724 | 1728 | 1739 |
 1743 | 1743 | 1749 | 1757 | 1758 | 1758 | 1761 | 1761 | 1766 | 1766 | 1774 | 1775 | 1780 | 1786 | 1795 | 1796 | 1797 | 179
8 | 1801 | 1802 | 1804 | 1806 | 1808 | 1808 | 1811 | 1813 | 1818 | 1818 | 1829 | 1829 | 1834 | 1839 | 1851 | 1857 | 1865 |
1868 | 1872 | 1877 | 1878 | 1881 | 1882 | 1885 | 1901 | 1902 | 1909 | 1909 | 1913 | 1918 | 1921 | 1924 | 1926 | 1933 | 1938
 | 1939 | 1948 | 1961 | 1968 | 1969 | 1970 | 1982 | 1990 | 1991 | 1991 | 1997 | 1998 |
------------------------------------------

 The time for parallel program is:              :0.002923 s
 The time for sequential program is:              :0.003265 s
[hamza@localhost application]$ 
```

Figure 4: Sorting a vector of 500 elements using the Quicksort method

execution time for the parallel solution is 0.00292s. The results show that the proposed parallel computing is very interesting for the large scale problems, where communication time is negligible in front of calculation time. Table 2 gives a first idea of the average acceleration rate.

–	Sequential solution	Parallel solution
15 elements	0.000015s	0.000458s
500 elements	0.00326s	0.00292s
Rate	0.46	15.68

Table 2: Average acceleration rate

Figure 5 represents two curves of the execution time according to the vector dimension to be sorted using the two programs (*sequential and parallel*). It is seen clearly that for the vectors with small size, the execution time of the sequential program is better than that of the parallel program, because the inter nodes communication time (*exchanged messages*) is significantly important relatively to calculus time. Starting from a dimension of 500 (*500 values*), we note a significant difference in favour of the parallel program as illustrated on table 2.

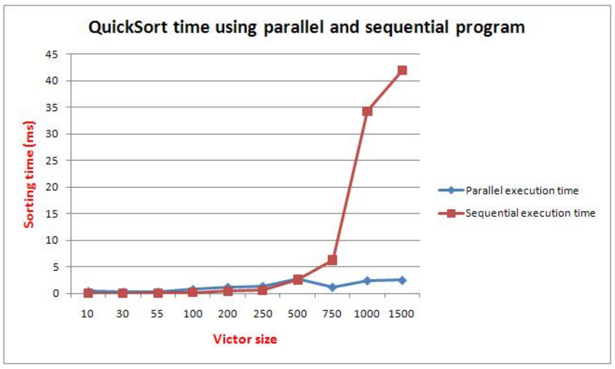


Figure 5: Execution time as function of the vector size

Figure 6 shows the execution time percent for both se-

[2]http://www.mcs.anl.gov/research/projects/mpi/

quential program and parallel program from total execution time as function of the sorted vector size. We can clearly observe that for complex problem; the execution time for parallel solution is very improved compared to that of the sequential one. This improves the scalability of the proposed solution. From figure 6, we can see also that when the vector size reaches 500, the parallel solution becomes more interesting.

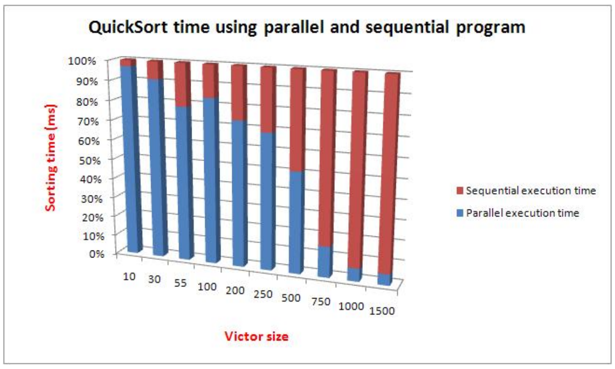


Figure 6: Percent of sequential and parallel execution times for sorting vector with the Quick sort method

Figure 7 shows the number of the generated messages du to the tree construction process (*for considering the free neighbouring nodes with higher capabilities*) as function of the degree of the Hanoi problem. When the degree of the Hanoi problem increase, the generated messages increase also. The most important number of messages is due to the diffusion tree construction process. When the diffusion tree is constructed, the number of overhead messages becomes stable.

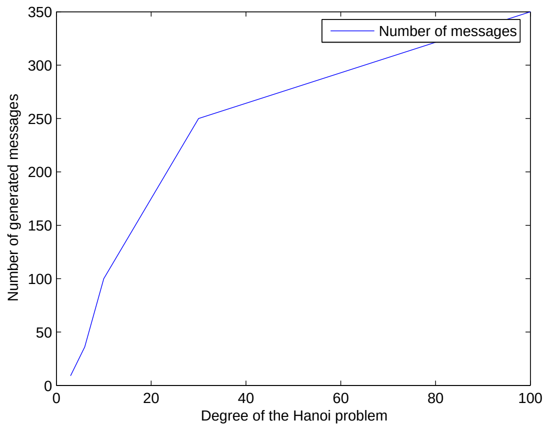


Figure 7: Number of generated messages as function of the degree of the Hanoi problem

5 Conclusions and Future Works

Master-Worker is a high-level programming framework that has been proposed to simplify the development of large scale parallel applications for computational grids. Many recent works are based on this concept. The hierarchical Master Worker paradigm is a very interesting approach to solve Divide and Conquer problem type , where the recursive problems are a good representative examples. The most difficult tasks are the fault tolerance, load balancing and scalability of the model.

In this paper, we have proposed a scalable hierarchical Master Worker model based on a dynamic on demand construction tree for tasks distribution. The solution is doubly generic; it can be used for any given parallel solutions of a recursive problem, but also it uses any existing P2P network as underlying architecture because P2P networks are a connected graphs. Each node in our model is both Master and Worker called "Must-Work".

By analyzing the various algorithms of the solution through validation examples, we can draw the following characteristics: the proposed solution minimizes the execution time in the case of great calculations, because parallelism allows the simultaneous execution of several tasks. It cures to the problems due to the recursivity, like the overflow of the stack as well as the capacity overshooting. It minimizes also the communication cost; each parent node assigns the calculation tasks directly to its children in its finger table (*using only one hop*). If a machine finishes its calculation, it will be released to carry out another calculation. In our model, the tasks are often affected to nodes with high capabilities.

The proposed solution guarantees the load balancing of the machines (*nodes*) in their affecting the tasks, since they are released according to the need of calculation. It thus minimizes the number of machines which participate at the same time on the resolution of the given recursive problem, because a child (*successor*) can be present in several finger tables, then since it will be released, it can be contacted by another parent (*predecessor*) for another task.

In terms of future works, first, we envision to test the proposed solution on a real specialized P2P platform, such as: Proactive[3] , XtremWeb[4] or JXTA[5]. Secondly, as opposed to Must-Work type node, we consider another node type which can refuses task execution, because it is belonging to an open P2P network (*contains free rider nodes*), we call it CAN-Work.

Acknowledgement

The authors would like to thank Mr A. Bendjoudi from CERIST and Miss W. Azzeghagh from University of Bejaia for their comments.

References

[1] C. Shirky, What is p2p..and what isn't, *O' Reilly Network*, 2001.

[3]http://proactive.inria.fr/
[4]http://www.xtremweb.net/
[5]https://jxta.dev.java.net/

[2] M. J. Flynn, Some computer organizations and their effectiveness, *IEEE Trans. Computers*, 21(9):948-960, 1972.

[3] K. Aida, W. Natsume and Y. F. Kata, Distributed Computing with Hierarchical Master-worker Paradigm for Parallel Branch and Bound Algorithm, *In Proceedings of the 3st International Symposium on Cluster Computing and the Grid (CCGRID)*, Tokyo, Japan, 2003.

[4] J.-P. Goux, S. Kulkarni, J. Linderoth and M. Yoder, “Master-Worker": An enabling framework for master-worker applications on the computational grid, *Cluster Computing*, Vol. 4, pp. 63-70, www.cs.wisc.edu/condor/doc/camera.doc, 2001.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, A scalable content addressable network, *in ACM SIGCOMM, New York*, 2001.

[6] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek and H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet application, *IEEE/ACM Transactions on networking*, Vol 11, No. 1, January 2003.

[7] B. Y. Zhao, J. Kubiatowich and A. D. Joseph, Tapestry: an infrastructure for fault-tolerant wide-area location and routing, *Technical report, No. UCB/CDS-01-1141*, Computer Science Division, University of California, Berkeley, April 2001.

[8] Z. Dai, F. Viale, X. Chi, D. Caromel and Z. Lu, Task-Based Fault-Tolerance Mechanism to Hierarchical Master/Worker with Divisible Tasks, *in Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications*, Seoul, Korea, 2009.

[9] I. Foster and C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, 1999.

[10] J. Pierre, G. J. Linderoth and M. Yoder, Metacomputing and the Master-Worker Paradigm, *ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Labroratory*, 2000.

[11] M. GhasemiGol, M. Sabzekar, H. Deldari and A. H. Bahmani, A Linda-based Hierarchical Master-Worker Model, *International Journal of Computer Theory and Engineering*, Vol. 1, No. 5, pp. 1793-8201, December, 2009.

[12] C. Banino, Scalability Limitations of the Master-Worker Paradigm for Grid Computing, *in proceedings of workshop on state-of-the-art in scientific computing (para’04), Denmark*, 2004.

[13] E. S. Manolakos and D. G. Galatopoulos *JavaPorts: An Environment to facilitate parallel computing on a heterogenous cluster of workstations*, Informatica, Vol. 23, pp. 97-105, 1999.

[14] J. Zhu, J. Gong, W. Liu, T. Song and J. Zhang, A collaborative virtual geographic environment based on P2P and Grids technologies , *Journal of Information Sciences*, Elsevier, Vol. 177, pp. 4621-4633, 2007.

[15] N. A. Al-Dmour and W. J. Teahan, ParCop: A Decentralized Peer-to-Peer Computing System, *In Proceedings of the ISPDC/HeteroPar’04, Cork, Ireland* , 2004.

[16] J. B. Ernst-Desmulier, J. Bourgeois, F. Spies and J. Verbeke, Adding New Features In A Peer-to-Peer Distributed Computing Framework, *in Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euromicro-PDP’05), Lugano, Switzerland* , 2005.

[17] J. Verbeke, N. Nadgir, G. Ruetsch and I. Sharapov, Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment, *In Proceedings of GRID 2002, Baltimore, Sun Microsystems, Inc.*, Palo Alto, CA 94303, USA, January 2002.

[18] I. Podnar, M. Rajman, T. Luu, F. Klemm and K. Aberer, Beyond Term Indexing: A P2P Framework forWeb Information Retrieval, *Informatica, Vol. 30, pp. 153-161, 2006.*

[19] A. J. Chakravarti, G. Baumgartner and M. Lauria, The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network, *IEEE Transactions on systems, man, and cybernetics*, Vol. 35, No. 3, may 2005.

[20] W. Wadge, Distributed Application Reliability on Unstable, Dynamic, P2P-based platforms, *CSAW, Kalkara, Malta, 2004.*

[21] M. Gupta, S. Mukhopadhyay and N. Sinha, Automatic Parallelization of Recursive Procedures, *Int. Journal of Par. Prog., Vol. 28(6):537-562, 2000.*

[22] M. Haveraaen, Efficient parallelisation of recursive problems using constructive recursion, *Euro-Par 2000 - Parallel Processing, volume 1900, Lecture Notes in Computer Science, Springer Verlag, 2000, pp. 758-761.*

[23] Gasper Fijavz, ColoringWeighted Series-Parallel Graphs, *Informatica Vol. 30, pp. 321-324, 2006.*