Time-stamp Incremental Checkpointing and its Application for an Optimization of Execution Model to Improve Performance of CAPE

Van Long Tran
Samovar, Télécom SudParis, CNRS, Université Paris-Saclay - 9 rue Charles Fourier, Évry, France
E-mail: van_long.tran@telecom-sudparis.eu and www.telecom-sudparis.eu
Hue Industrial College - 70 Nguyen Hue street, Hue city, Vietnam
E-mail: tvlong@hueic.edu.vn and www.hueic.edu.vn

Éric Renault
Samovar, Télécom SudParis, CNRS, Université Paris-Saclay - 9 rue Charles Fourier, Évry, France
E-mail: eric.renault@telecom-sudparis.eu and www.telecom-sudparis.eu

Viet Hai Ha
College of Education, Hue University - 34 Le Loi street, Hue city, Vietnam
E-mail: haviethai@gmail.com and www.dhsphue.edu.vn

Xuan Huyen Do
College of Sciences, Hue University - 77 Nguyen Hue street, Hue city, Vietnam
E-mail: doxuanhuyen@gmail.com and www.husc.edu.vn

Keywords: OpenMP, OpenMP on cluster, CAPE, Checkpointing-Aided Parallel Execution, Checkpointing, Incremental checkpointing, DICKPT, TICKPT

Received: March 29, 2018

CAPE, which stands for Checkpointing-Aided Parallel Execution, is a checkpoint-based approach to automatically translate and execute OpenMP programs on distributed-memory architectures. This approach demonstrates high-performance and complete compatibility with OpenMP on distributed-memory systems. In CAPE, checkpointing is one of the main factors acted on the performance of the system. This is shown over two versions of CAPE. The first version based on complete checkpoints is too slow as compared to the second version based on Discontinuous Incremental Checkpointing. This paper presents an improvement of Discontinuous Incremental Checkpointing, and a new execution model for CAPE using new techniques of checkpointing. It contributes to improve the performance and make CAPE even more flexible.

Povzetek: Predstavljena je izboljšava CAPE - paralelno izvajanje, usmerjeno s podporo redundance.

1 Introduction

In order to minimize programmers' difficulties when developing parallel applications, a parallel programming tool at a higher level should be as easy-to-use as possible. MPI [1], which stands for Message Passing Interface, and OpenMP [2] are two widely-used tools that meet this requirement. MPI is a tool for high-performance computing on distributed-memory environments, while OpenMP has been developed for shared-memory architectures. If MPI is quite difficult to use for non programmers, OpenMP is very easy to use, requesting the programmer to tag the pieces of code to be executed in parallel.

Some efforts have been made to port OpenMP on distributed-memory architectures. However, apart from our solution, no solution successfully met the two following requirements: 1) to be fully compliant with the OpenMP standard and 2) high performance. Most prominent approaches include the use of an SSI [3], SCASH [4], the use of the RC model [5], performing a source-to-source translation to a tool like MPI [6, 7] or Global Array [8], or Cluster OpenMP [9].

Among all these solutions, the use of a Single System Image (SSI) is the most straightforward approach. An SSI includes a Distributed Shared Memory (DSM) to provide an abstracted shared-memory view over a physical distributed-memory architecture. The main advantage of this approach is its ability to easily provide a fully-compliant version of OpenMP. Thanks to their shared-memory nature, OpenMP programs can easily be compiled and run as processes on different computers in an SSI. However, as the shared memory is accessed through the network, the synchronization between the memories involves an important overhead which makes this approach hardly scalable. Some experiments [3] have shown that the larger the number of threads, the lower the performance. As a result, in order to reduce the execution time overhead involved by the use of an SSI, other approaches have been proposed. For example, SCASH only maps the shared variables of the processes onto a shared-memory area at-

tached to each process, the other variables being stored in a private memory, and the RC model uses the relaxed consistency memory model. However, these approaches have difficulties to identify the shared variables automatically. As a result, no fully-compliant implementation of OpenMP based on these approaches has been released so far. Some other approaches aim at performing a source-to-source translation of the OpenMP code into a MPI code. This approach allows the generation of high-performance codes on distributed-memory architectures. However, not all OpenMP directives and constructs can be implemented. As yet another alternative, Cluster OpenMP, proposed by Intel, also requires the use of additional directives of its own (ie. not included in the OpenMP standard). Thus, this one cannot be considered as a fully-compliant implementation of the OpenMP standard either.

Concerning to bypass these limitations, we developed CAPE [10, 15] which stands for Checkpointing-Aided Parallel Execution. CAPE is a solution that provides a set of prototypes and frameworks to automatically translate OpenMP programs for distributed memory architectures and make them ready for execution. The main idea of this solution is using incremental checkpoint techniques (ICKPT) [11, 12] to distribute the parallel jobs and their data to other processes (the fork phase of OpenMP), and collect the results after the execution of the jobs from all processors (the join phase of OpenMP). ICKPT is also used to deal with the exchange of shared data automatically.

Although CAPE is still under development, it has shown its ability to provide a very efficient solution. For example, a comparison with MPI showed that CAPE is able to reach up to 90% of the MPI performance [13, 14]. This has to be balanced with the fact that CAPE for OpenMP requires the introduction of few `pragma` directives only in the sequential code, i.e. no complex code from the user point of view, while writing a MPI code might require the user to completely refactorise the code. Moreover, as compared to other OpenMP for distributed-memory solutions, CAPE is fully compatible with OpenMP [13, 15].

This paper presents an improvement of DICKPT – a checkpoint technique for CAPE, and a new execution model applied these new checkpoints, that improves the performance and the flexibility of CAPE. A part of these results were also presented and published at the SoICT's 2017 conference [16]. The paper is organized as follows: the next section describes CAPE mechanism, capabilities and restrictions in details. Section 3 presents a development of checkpointing that are used in CAPE. Then, Section 4 presents the design and the implementation of the new execution model based on the new checkpointing techniques. The analysis and evaluation of both new checkpointing and execution model are presented in Section 5. Section 4 shows the result of the experimental analysis. Finally, Section 5 concludes the paper and presents our future works.

2 CAPE principles

In order to execute an OpenMP program on distributed-memory systems, CAPE uses a set of templates to translate an OpenMP source code into a CAPE source code. Then, the generated CAPE source code is compiled using a traditional C/C++ compiler. At last, the binary code can be executed independently on any distributed-memory system supporting the CAPE framework. The different steps of the CAPE compilation process for C/C++ OpenMP programs is shown in the Figure 1.

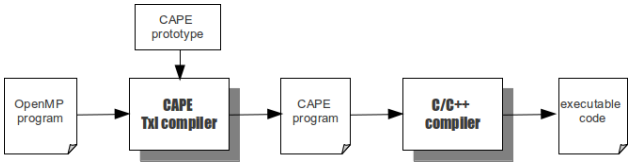

Figure 1: Translation of OpenMP programs with CAPE.

2.1 Execution model

The CAPE execution model is based on checkpoints that implement the OpenMP fork-join model. This mechanism is shown in Figure 2. To execute a CAPE code on a distributed-memory architecture, the program first runs on a set of nodes, each node being run as a process. Whenever the program meets a parallel section, the master node distributes the jobs among the slave processes using the Discontinuous Incremental Checkpoints (DICKPT) [12, 13] mechanism. Through this approach, the master node generates DICKPTs and sends them to the slave nodes, each slave node receives a single checkpoint. After sending checkpoints, the master node waits for the results to be returned from the slaves. The next step is different depending upon the nature of the node: the slave nodes receive their checkpoint, inject it into their memory, execute their part of the job, and sent back the result to the master node by using DICKPT; the master node waits for the results and after receiving them all, merges them before injection into its memory. At the end of the parallel region, the master sends the resulting checkpoint to every slaves to synchronize the memory space of the whole program.

2.2 Translation from OpenMP to CAPE

In the CAPE framework, a set of functions has been defined and implemented to perform the tasks devoted to DICKPT, typically, distributing checkpoints, sending/receiving checkpoints, extracting/injecting a checkpoint from/to the program's memory, etc. Besides, a set of templates has been defined in the CAPE compiler to perform the translation of the OpenMP program into the CAPE program automatically and make it executable in the CAPE framework. So far, nested loops and shared-data variable constructs are not supported yet. However, this is not regarded as an issue as this can be solved at the level

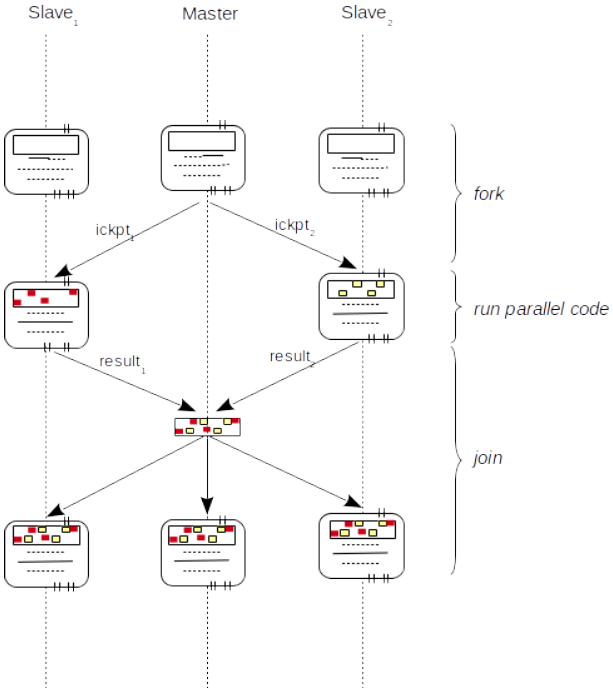

Figure 2: CAPE execution model.

of the source-to-source translation and does not require any modifications in the CAPE philosophy. In this end, CAPE can only be applied to OpenMP programs matching the Bernstein's conditions [17].

After the translations operated by the CAPE compiler, the OpenMP source code is free of any OpenMP directives and structures. Figure 3 presents an example of code substitution for the specific case of the `parallel for` construct. This example is typical of those we implemented for the other constructs [7]. The automatically generated code is based on the following functions that are part of the CAPE framework:

- `start( )` sets up the environment for the generation of DICKPTs.
- `stop( )` restores the environment used for the generation of DICKPT.
- `create(file)` generates a checkpoint with name `file`.
- `inject(file)` injects a checkpoint into the memory of the current process.
- `send(file, node)` sends a checkpoint from the current process to another.
- `wait_for(file)` waits for checkpoints and merges them to create another one.
- `merge(file1,file2)` merges two checkpoints together.

```
# pragma omp parallel for
  for ( A ; B ; C )
      D ;
```

↓ automatically translated into ↓

```
1 if ( master ( ) )
2    start ( )
3    for ( A ; B ; C )
4         create ( before )
5         send ( before, slave_x )
6    create ( final )
7    stop ( )
8    wait for ( after )
9    inject ( after )
10   if ( ! last parallel ( ) )
11           merge ( final, after )
12           broadcast ( final )
13 else
14   receive ( before )
15   inject ( before )
16   start ( )
17   D
18   create ( after_i )
19   stop ( )
20   send ( after_i, master )
21   if ( ! last parallel ( ) )
22           receive ( final )
23           inject ( final )
24   else
25           exit
```

Figure 3: Template for the `parallel for` with incremental checkpoints.

- `broadcast(file)` sends a checkpoint to all slave nodes.
- `receive(file)` waits for and receives a checkpoint.

2.3 Discontinuous incremental checkpointing on CAPE

Checkpointing is the technique that saves the images of a process at a point during its lifetime, and allows it to be resumed at the saving's time if necessary [11, 18]. Using checkpointing, processes can resume their execution from a checkpoint state when a failure occurs. So, no need to take time to initialize and execute it from the begin. These techniques are introduced since two decades ago. Nowadays, they are researched and used widely on fault-tolerance, applications trace/debugging, roll-back/animated playback, and process migration.

Basically, checkpointing techniques can be categorized

into two groups: completed checkpoints and incremental checkpoints. Completed checkpointing [18, 19, 20] saves all information regarding the process at the points that it generate checkpoints. The advantages of this technique is reducing the time of generation and restoration. However, the checkpoint's size is too large. Incremental checkpointing [11, 21, 22, 23, 12, 24] only saves the modified information as compared to the previous checkpoint. This technique reaches advantages of reducing checkpoint's overhead and checkpoint's size, so it is in widely used in distributed computing. Besides, using data compression to reduce checkpoint's size [11, 21, 24], it is also focus on the techniques that detect modified data but reach the minimum of size. Some typical techniques are using page-based protection to identify the pages in memory that have been modified [11, 22, 23], using word-level granularity [21, 12], using block encoding [22], using user-directed and memory exclusion [11], using live variable analysis [24].

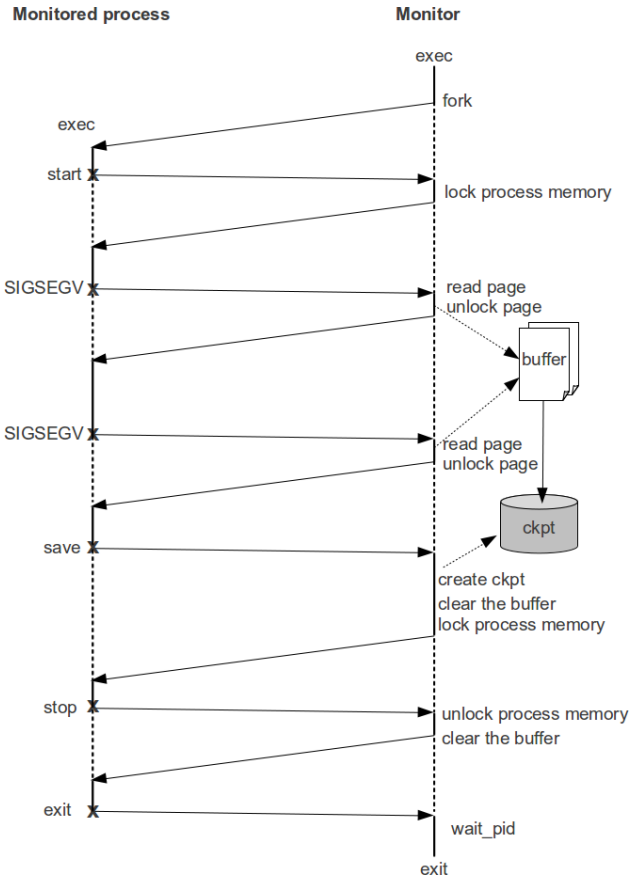

Figure 4: Principle of DICKPT in cases of checkpointing.

In CAPE, Discontinuous Incremental Checkpointing (DICKPT) is a development based on incremental checkpointing, that contains two kinds of data, register information and modified data of the process. In which, the first one is copied from all register data of the process, and the second one is identified based on write-protection techniques.

Figure 4 shows the steps to monitor and generate a checkpoint of a process on CAPE. It is done by an other process making use of the $ptrace$ Unix system call. The idea of these steps is that, at the beginning of the parallel region, the monitor sets all page of monitor process at write-protected. Whenever the monitored process wants to write into any write-protected page, a $SIGSEGV$ signal is generated. Then, the monitor saves the data of this page, removes the write-protection and lets the monitored process write into the page. At the end of the region, the monitor compares the saved data with the current data of monitored process page. The difference are extracted and saved into checkpoint file.

2.4 Remarks

The good performance of CAPE as compared to those of MPI and the full compliance to the OpenMP specifications [13, 15, 14] have made CAPE a good alternative to port OpenMP on distributed-memory architectures. So far, the implementation of CAPE is not complete, some disadvantages can be listed:

1. DICKPT saves all modified data of process, including temporary and private variables. This is an unnecessary synchronization inside an OpenMP program.

2. As shown in Figure 2, the master node might act as a bottleneck while waiting for checkpoints from the salves, merging checkpoints and/or sending back data to slaves for memory synchronization.

3. To distribute jobs to slaves, the master node generates a number of checkpoints that depends upon the number of slave nodes and so that each slave node receives a checkpoint (see Figure 7). This method can reach a high-level of optimization. However, it might not be enough flexible for some cases like 1) the number of slaves may not be identified at compile time, 2) the OpenMP source code should be modified to detect when the master generates the checkpoint and 3) the dynamic scheduling of OpenMP cannot be implemented using this method.

4. After distributing the jobs, the slave nodes execute the divided jobs while the master does nothing until the reception of the resulting checkpoints from the slaves, which clearly wastes resources.

5. For synchronization, the checkpoints should be sent by order in order to resume exactly the last state of process.

3 Time-stamp incremental checkpointing (TICKPT)

Time-stamp Incremental Checkpointing (TICKPT) is an improvement of DICKPT by adding new factor – time-stamp – into incremental checkpoints and by removing un-

necessary data based on data-sharing variable attributes of OpenMP program.

Basically, TICKPT contains three mandatory elements including register's information, modified region in memory of the process, and their time-stamp. As well as DICKPT, in TICKPT, the register's information are extracted from all registers of the process in the system. However, the time-stamp is added to identify the order of the checkpoints in the program. This contributes to reduce the time for merging checkpoints and selecting the right element if located at the same place in memory. In addition, only the modified data of shared variables are detected and saved into checkpoints. It makes checkpoint's size significantly reduced depending on the size of private variables of the OpenMP program.

To present the order of checkpoints in a program, time-stamps have to represent the order of the instructions when it is executed. For the general case, an activation tree [25] can be used to identify the sequence of function call in a program. For CAPE, checkpoints are always generated in same level of functions, so that the program counter can be used to ensure simplicity. However, if the instruction is a `loop`, the program counter is combined with the loop iteration to represent the order of the loop exactly.

To detect modified data, the write-protection mechanism is used. However, only the shared variables are written down in the checkpoint file. The matter in here is how to detect private and shared variables.

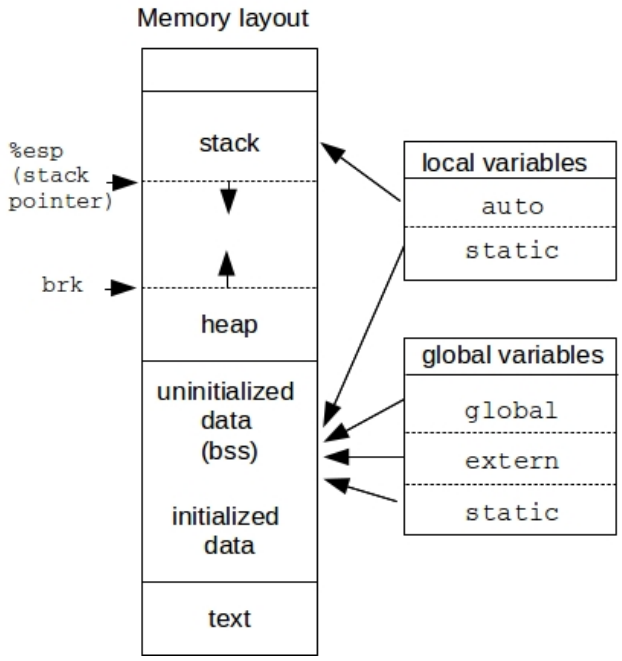

Figure 5: Allocation of OpenMP program's variables in virtual process memory.

In an OpenMP program, data-sharing variable attributes can be set up either, implicitly or explicitly [2]. All variables declared outside an `#pragma omp parallel` directive are implicitly shared. This includes all global and `static` local variables allocated in `heap` and `data` segments of the process's memory, and local variables allocated on the `stack` (see Figure 5). The variables in `heap` and `data` segments can easily be identified by their address. For the variables on the `stack`, we save the stack pointer before entering the `#pragma omp parallel` region. Variables declared before the stack pointer are shared. The others, are private.

To explicitly, change the status of a variable, the programmer can use data-sharing attributes like OpenMP directive `#pragma omp threadprivate (list of variables)` and relative clauses. The OpenMP data-sharing clauses are shown in Table 1.

Clauses	Description
default(none|shared)	Specifying the default behavior of variables
shared(list)	Specifying the list of shared variables
private(list)	Specifying the list of private variables
firstprivate(list)	Allowing to access value of the list of private variables in the first time
lastprivate(list)	Allowing to share value of the list of private variables at the end of parallel region
copyin(list)	Allowing to access value of threadprivate variables
copyprivate(list)	Specifying the list of private variables that should be shared among all threads.
reduction(list, ops)	Specifying the list of variables that are subject to a reduction operation at the end of the parallel region.

Table 1: OpenMP data-sharing clauses.

4 A new execution model for CAPE

In order to improve the performance of CAPE and its flexibility, we designed a new execution model that extends the one presented in Section 2.1. In this new execution model, DICKPT is replaced by TICKPT. Figure 6 illustrates the model which can be described as follows:

1. At the beginning of the program, all nodes in the system execute the same sequential code.

2. When a parallel region is reached, the master process creates a set of incremental checkpoints. The number of incremental checkpoints depends upon the number of tasks in the parallel region. Each incremental checkpoint contains the state of the program to be

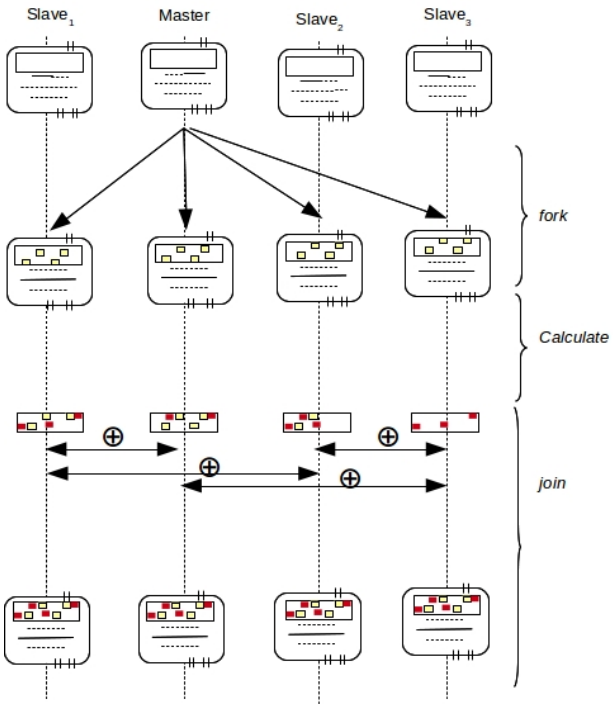

Figure 6: The new execution model for CAPE.

used to resume its execution in another process at the saved time.

3. The master process scatters the set of incremental checkpoints. Each node receives some of the checkpoints generated by the master process. This step is illustrated in the Figure 8.

4. The received incremental checkpoints are injected into the slave processes' memories.

5. The slave processes resume their execution.

6. Results on slave processes are extracted by identifying the modified regions and saved as an incremental checkpoint.

7. Incremental checkpoints of each process is sent back to the master node. Incremental checkpoints are combined altogether to generate a single checkpoint. This step can be distributed among the processes if need be.

8. The final combined incremental checkpoint is injected in the master process' memory and the master process can resume its execution.

Changing the execution model implies changing the translation templates. Figure 9 presents the template for the `#pragma omp parallel for` directive that adapts to the new execution model. The other OpenMP directives can be designed in a similar way. For this template, CAPE operates as follows:

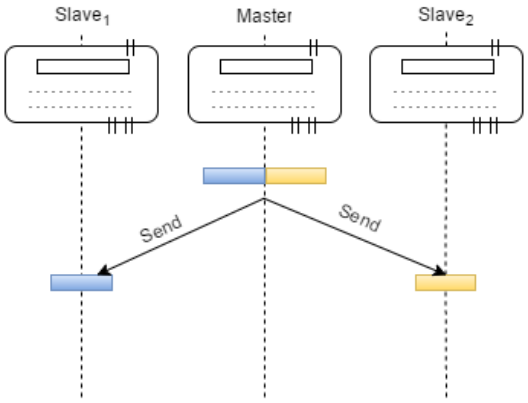

Figure 7: Scheduling method in CAPE-2.

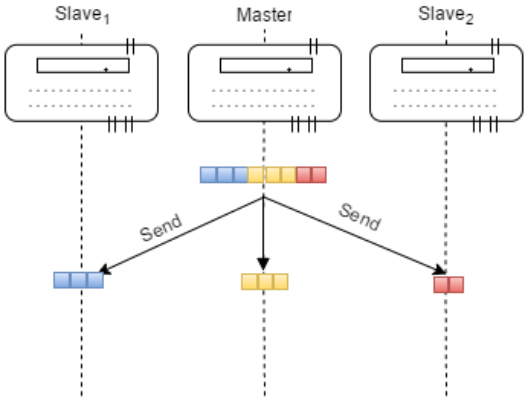

Figure 8: Scheduling method with the new execution model.

– $generate\_dickpt(before_i)$ (line 3): at each loop iteration, the master process generates an incremental checkpoint.

– $scatter(before,\ \&recv_n,\ master)$ (line 4): the master process scatters the checkpoints to the available processes, including itself. Each process receives some of the checkpoints ($recv_n$).

– $inject(recv_n)$ (line 5): each checkpoint is injected into the target process' memory.

– the execution is resumed on instruction D (line 6).

– $generate\_dickpt(after_n)$ (line 7): each process generates an incremental checkpoint that saves the result of its execution.

– $allreduce(after_n,\ \&after,\ [< ops >])$ (line 8): the $after_n$ checkpoint of process n is sent to the other processes. Checkpoints are combined, calculated and saved in a new $after$ checkpoint. With TICKPT, the order of checkpoints is presented in each of them, so this is performed using the Recursive Doubling algorithm [26] as illustrated in Figure 10.

– $inject(after)$ (line 9): incremental checkpoint $after$

```
# pragma omp parallel for
  for ( A ; B ; C )
      D ;
```

↓ automatically translated into ↓

```
1 if ( master ( ) )
2   for (A ; B ; C)
3     generate_dickpt (before_i)
4 scatter(before, &recv_n ,master)
5 inject(recv_n )
6 D
7 generate_dickpt (after_n)
8 allreduce(after_n,&after,[<ops>])
9 inject(after)
```

Figure 9: Prototype for the `parallel for` with the new execution model.

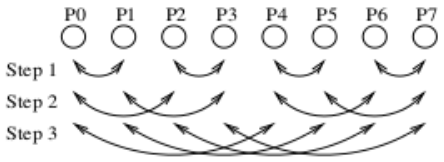

Figure 10: Recursive doubling for allreduce.

is injected into the application's memory to synchronize the state of the program on all nodes.

5 Analysis and evaluation

5.1 From DICKPT to TICKPT

As presented in Section 3, TICKPT is an evolution of DICKPT. It creates and adds time-stamps into checkpoints to make them more flexible and to reduce synchronization time when applying on CAPE. In addition, it removes unnecessary data to reduce checkpoint's size. For the synchronization time, we will analyse and evaluate the whole performance of CAPE. For checkpoint's size, we consider the amount of the modified data generated by TICKPT and DICKPT after having executed the piece of code in Figure 11 in each node, with various values for N.

Data contained in A, B and C variables are changed. The DICKPT counts them all, while TICKPT only counts data in variable C. Therefore, the amount of modified data significantly reduced with TICKPT as shown in Figure 12.

5.2 Analysis of the new execution model

Moving from a scheduling of CAPE processes based on the number of nodes (Figure 7) to a scheduling based on

```
#define N 1000
...
int A[N], B[N], C[N], i;
...
#pragma omp parallel for
                private(A,B) shared(C)
for(i = 0; i <N; i++){
       A[i] = i;
       B[i] = N - i;
       C = A[i] + B[i] ;
}
```

Figure 11: A piece of OpenMP code used to consider the amount of modified data with the two checkpoint techniques.

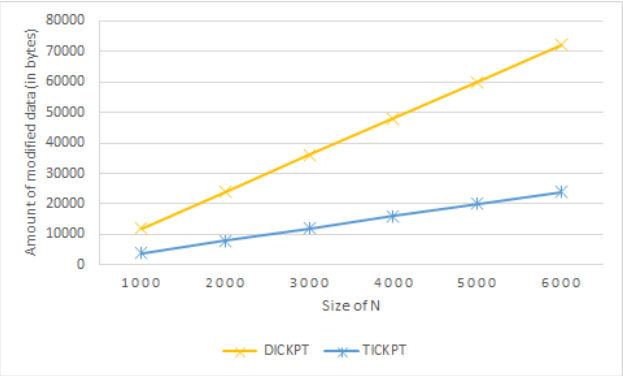

Figure 12: Amount of modified data (in bytes) generated by both methods.

the number of jobs (Figure 8) makes CAPE more flexible at least for the three following reasons:

1. The number of available processes can be identified at runtime. The master node can distribute the jobs to all available processes.

2. All OpenMP scheduling mechanisms such as `static` and `dynamic` can be implemented on CAPE. This is because the master node generates a number of checkpoints depending on the number of jobs. First step, one checkpoint can be sent to each slave node to execute a divided job. When the slave node finishes, it is sent the next checkpoint that has not been executed yet.

3. There is no need to modify the OpenMP source code to detect the location of the master process that generated the incremental checkpoints and sent them to the slave nodes.

For performance analysis and evaluation, considering that both initial and sequential codes of the program are executed in the same way in any processes of the system, only the execution time of the parallel regions has been considered.

Let t_f be the execution time of the fork phase, t_c be the computation time to execute the divided jobs and t_j be the

time for the join phase, ie. the time to synchronize data after executing the divided jobs at all nodes. For each parallel region, the execution time can be computed using equation 1.

$$t = t_f + t_c + t_j \tag{1}$$

Note that t_f is similar for both methods as both consider the work-shared steps and the generation of incremental checkpoints, and an incremental checkpoint only consist of very few bytes, ie. the time for the fork phase is close to zero.

In the previous execution model, the master process was not involved in the computation phase, and resources ware wasted. Let n be the number of jobs and p be the number of processes. Assume that each process takes one unit of time to execute a job, and the number of jobs is equally divided equally among the processes. The value for t_c becomes:

$$t_c = \left\lceil \frac{n}{p-1} \right\rceil \tag{2}$$

With the new execution model, all processes are involved in the computation phase so that t_c is equal to:

$$t_c = \left\lceil \frac{n}{p} \right\rceil \tag{3}$$

t_j is also impacted by the new execution model. In the previous model, the value for t_j is equal to the time for the slave processes to send their results to the master node for combination plus the time to receive the final checkpoint and inject it into the process' memory. This work is done sequentially. Thus, the time to send or receive a checkpoint is given by:

$$t_j = 2(p-1) \tag{4}$$

With the new execution model, the Recursive Doubling algorithm [26] is applied to communicate between all processes, so that t_j becomes:

$$t_j = \lceil log_2(p) \rceil \tag{5}$$

Computation time t_c is the most important factor that affects the execution time of a parallel region. From equations (2) and (3), it is easy to demonstrate that t_c for the previous execution model is always larger than t_c for the new execution model, ie. the execution time for CAPE is reduced with this new execution model. And the resources are used more efficiently.

Besides, the use of the Recursive Doubling algorithm during the join phase with the new execution model allows saving time when synchronizing data between processes. This is highlighted by comparing equations (4) and (5) with the previous execution model and the new execution model respectively.

6 Experiments

In order to measure the impact of the new execution model on the performance, as mathematically analyzed in Section 5, some experiments were conducted. These experiments were performed on 4-node and 16-node clusters. Each node includes an Intel core i3-2100, a dual-core 4-thread CPU running at 3.10 GHz and 2 GB of RAM. These computers are connected using a 100 Mb/s Ethernet network. To avoid external influence as much as possible, the entire system was dedicated to the tests during all of the performance of the measurement campaign.

The program used as the basis for these experiments is the classic matrix-matrix multiplication. The sizes of the matrices are increased from 1600×1600, 3200×3200 to 9600×9600. Each program is executed at least 10 times to measure the total execution times and a confidence interval of at least 98% has been always achieved for the measures. Data reported here are the means of the 10 measures.

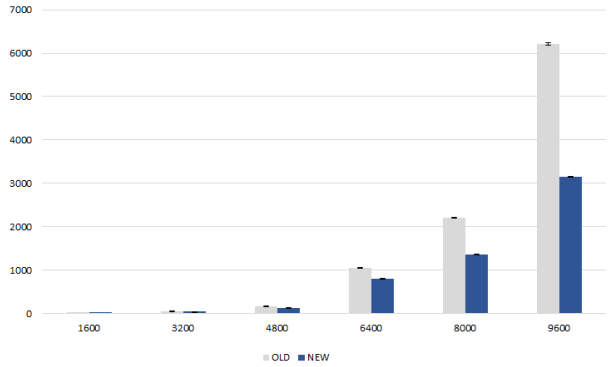

Figure 13: Total execution time (in seconds) for both models on 4-node clusters.

Figure 13 and 14 present the total execution time of CAPE on 4-node and 16-node clusters respectively. As can be seen from these figures, the execution time of both models are shown, the gray color (OLD) represents the previous execution model, and the blue one (NEW) represents the new execution model. The horizontal axis shows the size of the matrix and the vertical axis shows the execution time in seconds.

For the 4-node cluster, as compared with the previous model, the execution time of the new model is reduced significantly and the reduction is inversely proportional to the size of the matrix. The larger the size of the matrix, the shorter in time. This is due to the fact that there are only three nodes executing the divided jobs on the previous execution model. The master just divides and distributes jobs to slaves, and then waits for the results return. It does not participate in the computational part. In contrast, with the new execution model, master node receives and executes a part of the divided jobs. Therefore the computation time (t_c) on this model is much lower than on previous model, especially on the cluster with only 4 nodes.

For the 16-node cluster, the result in Figure 14 shows the

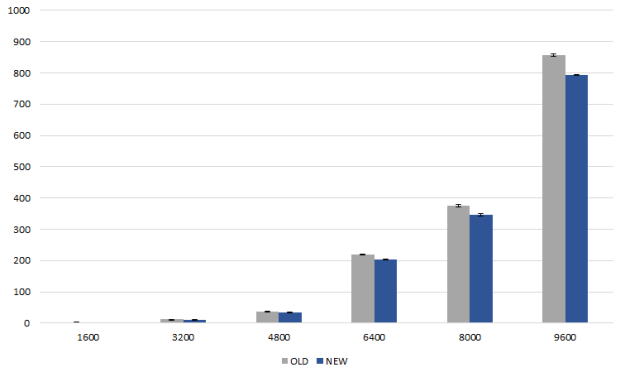

Figure 14: Total execution time (in seconds) of two models on 16-node clusters.

same trend, ie. the new execution model is better than the previous one. However, the distance between the execution time of both models is closer. It maintains a saving time around 10%. This is due to the larger number of nodes that leads to less time to compute the divided jobs. Therefore, the total saving time of t_c in this case is smaller.

Figure 15 presents the execution time of the fork (t_f), computation (t_c) and join (t_j) phases for both previous and new execution models on the master node with the 16-node cluster. The matrix size 9600 $\times$ 9600 is selected in this case.

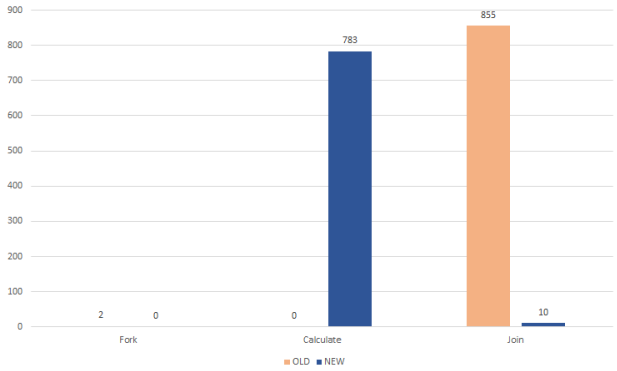

Figure 15: Execution time (in seconds) fork, computation and join phases for both previous and new execution models on the master node.

In the previous model, after the fork operation, the master node waits for the results from the slave nodes. Therefore, the value for t_c is equal to zero. For the same phase using the new execution model, the master node participates in the execution together with the slave nodes, so that t_c is much larger than zero. However, the new execution model uses the free resources of the master node to compute a part of the jobs of the whole program. This does not increase the whole execution time, but also contributes to improve the global efficiency of the system. The join phase comes right after the computation phase in the previous execution model. At this time, the master node waits for the results from the slave nodes and the synchronization of data. With the new execution model, this time is dedicated to the synchronization of data. Therefore, t_j is much smaller for the new execution model as compared with the previous execution one.

Indeed, both the theoretical analysis and the practical experiments on clusters composed of 4 nodes and 16 nodes when comparing the previous execution model and the new execution model show that the resources of the system are used more efficiently and the execution time is significantly reduced (decreased at least by 10%). This shows that the new execution model is a good direction to pursue the development of CAPE in the future.

7 Conclusion and future works

In this paper, we presented and analysed the disadvantages of previous version of CAPE. We also proposed a new method name TICKPT to improve the previous checkpointing technique. Then, TICKPT is applied to improve the previous execution model. Both theoretical analysis and experimentation showed that checkpoint's size and risk of bottlenecks in execution model are reduced significantly while the performance and the flexibility of CAPE are improved.

For the future, we will keep on developing CAPE using this new execution model. We will also try to determine how to combine checkpoints more efficiently to implement OpenMP's shared-data environment variables.

Acknowledgement

We thank the technical program committee members of SoICT 2017 conference for their reviews and comments that greatly improved the manuscript. We also thank Prof. Zhenjiang Hu, chair of the Software Engineering section for the discussion at the conference, that provided us ideas to extend this article.

References

[1] Message Passing Interface Forum (2014) MPI: A Message-Passing Interface Standard, `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[2] OpenMP ARB (2013) OpenMP application program interface version 4.0, `http://www.openmp.org`.

[3] Morin, Christine and Lottiaux, Renaud and Vallée, Geoffroy and Gallard, Pascal and Utard, Gaël and Badrinath, Ramamurthy and Rilling, Louis (2003) Kerrighed: a single system image cluster operating system for high performance computing, *Euro-Par 2003 Parallel Processing*, Springer,

pp. 1291–1294. https://doi.org/10.1007/978-3-540-45209-6_175.

[4] Sato, Mitsuhisa and Harada, Hiroshi and Hasegawa, Atsushi and Ishikawa, Yutaka (2001) Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system, *Scientific Programming*, Hindawi, pp. 123–130. http://doi.org/10.1155/2001/605217.

[5] Karlsson, Sven and Lee, Sung-Woo and Brorsson, Mats (2002) A fully compliant OpenMP implementation on software distributed shared memory, *High Performance Computing—HiPC 2002*, Springer, Berlin, pp. 195–206. https://doi.org/10.1007/3-540-36265-7_19.

[6] Basumallik, Ayon and Eigenmann, Rudolf (2005) Towards automatic translation of OpenMP to MPI, *Proceedings of the 19th annual international conference on Supercomputing (SC)*, ACM, pp. 189–198. https://doi.org/10.1145/1088149.1088174.

[7] Dorta, Antonio J and Badía, José M and Quintana, Enrique S and de Sande, Francisco (2005) Implementing OpenMP for clusters on top of MPI , *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer, pp. 148–155. https://doi.org/10.1007/11557265_22.

[8] Huang, Lei and Chapman, Barbara and Liu, Zhenying (2005) Towards a more efficient implementation of OpenMP for clusters via translation to global arrays, *Parallel Computing*, Elsevier, pp. 1114–1139. https://doi.org/10.1016/j.parco.2005.03.015.

[9] Hoeflinger, Jay P (2006) Extending OpenMP to clusters, *White Paper*, Intel Corporation.

[10] Renault, Éric (2007) Distributed Implementation of OpenMP Based on Checkpointing Aided Parallel Execution, *A Practical Programming Model for the Multi-Core Era*, Springer, pp. 195–206. https://doi.org/10.1007/978-3-540-69303-1_22.

[11] Plank, James S and Beck, Micah and Kingsley, Gerry and Li, Kai (1994) Libckpt: Transparent checkpointing under unix, *White Paper*, Computer Science Department. https://pdfs.semanticscholar.org/bd21/4d6a94edf9bd4f97b4467c545dafd8138e8a.pdf.

[12] Ha, Viet Hai and Renault, Éric (2011) Discontinuous Incremental: A new approach towards extremely lightweight checkpoints, *Computer Networks and Distributed Systems (CNDS)*, IEEE, pp. 227–232. https://doi.org/10.1109/CNDS.2011.5764578.

[13] Ha, Viet Hai and Renault, Eric (2011) Design and performance analysis of CAPE based on discontinuous incremental checkpoints, *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, IEEE, pp. 862-867. https://doi.org/10.1109/PACRIM.2011.6033008.

[14] Tran, Van Long and Renault, Eric and Ha, Viet Hai (2016) Analysis and evaluation of the performance of CAPE, *IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress*, IEEE, pp. 620–627. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0104.

[15] Ha, Viet Hai and Renault, Eric (2011) Improving performance of CAPE using discontinuous incremental checkpointing, *High Performance Computing and Communications (HPCC)*, IEEE, pp. 802–807. https://doi.org/10.1109/HPCC.2011.114.

[16] Tran, Van Long and Renault, Éric and Do, Xuan Huyen and Ha, Viet Hai (2017) Design and implementation of a new execution model for CAPE, *Proceedings of the Eighth International Symposium on Information and Communication Technology (SoICT's 2017)*, ACM, pp. 453–459. https://doi.org/10.1145/3155133.3155199.

[17] Bernstein (1966) Analysis of Programs for Parallel Processing, *IEEE Transaction on Electronic Computers*, IEEE, pp. 757–763. https://doi.org/10.1109/PGEC.1966.264565.

[18] Cores, Iván and Rodríguez, Mónica and González, Patricia and Martín, María J (2016) Reducing the overhead of an MPI application-level migration approach, *Parallel Computing*, Elsevier, pp. 72–82. https://doi.org/10.1016/j.parco.2016.01.012.

[19] Li, C-CJ and Fuchs, W Kent (1990) Catch-compiler-assisted techniques for checkpointing, *Fault-Tolerant Computing (FTCS)*, IEEE, pp. 74–81. https://doi.org/10.1109/FTCS.1990.89337.

[20] Chen, Zhengyu and Sun, Jianhua and Chen, Hao (2016) Optimizing Checkpoint Restart with Data Deduplication, *Scientific Programming*, Hindawi. https://doi.org/10.1155/2016/9315493.

[21] Plank, James S and Xu, Jian and Netzer, Robert HB (1995) Compressed differences: An algorithm for fast

incremental checkpointing, *Technical Report CS-95-302*, University of Tennessee.

[22] Hyochang, NAM and Jong, KIM and Hong, Sung Je and Sunggu, LEE (1997) Probabilistic checkpointing, *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, IEEE. https://doi.org/10.1109/FTCS.1997.614077.

[23] Mehnert-Spahn, John and Feller, Eugen and Schoettner, Michael (2009) Incremental checkpointing for grids, *Proceedings of the Linux Symposium*, Montreal, Quebec, Canada, pp. 201–220.

[24] Cores, Iván and Rodríguez, Gabriel and González, Patricia and Osorio, Roberto R and others (2013) Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes, *New Generation Computing*, Springer, pp. 163–185. https://doi.org/10.1007/s00354-013-0302-4.

[25] Alfred, V.Aho and Monica, S. Lam and Ravi, Sethi and Jeffrey, D. Ullman (2006) *Compilers Principles, Techniques,& Tools*, Addion Wesley.

[26] Thakur, Rajeev and Rabenseifner, Rolf and Gropp, William (2005) Optimization of collective communication operations in MPICH, *International Journal of High Performance Computing Applications*, Sage Publications, pp. 49–66. https://doi.org/10.1177/1094342005051521.