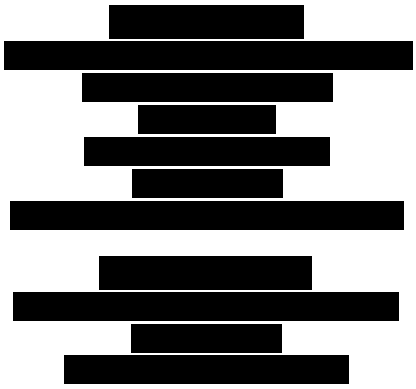
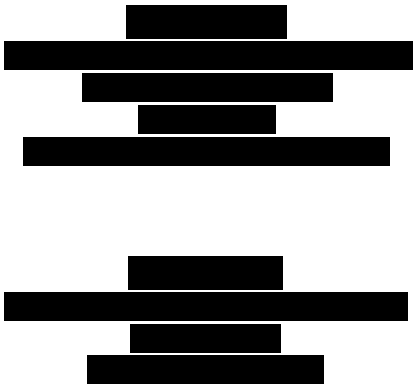
Design and implementation of a new execution model for CAPE





ABSTRACT

CAPE, which stands for Checkpointing-Aided Parallel Execution, is an approach based on checkpoints to automatically translate and execute OpenMP programs on distributed-memory architectures. This approach demonstrates high-performance and completes compatibility with OpenMP on distributed-memory system. This paper presents a new design and implementation model for CAPE that improves the performance and makes CAPE even more flexible.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Parallel programming languages**; • **Software and its engineering** → *Development frameworks and environments*;

KEYWORDS

CAPE, Checkpointing Aided Parallel Execution, OpenMP, Parallel Programming, Distributed Computing, HPC

ACM Reference Format:
Van Long Tran, Éric Renault, Xuan Huyen Do, and Viet Hai Ha. 2017. Design and implementation of a new execution model for CAPE. In *SoICT '17: Eighth International Symposium on Information and Communication Technology, December 7–8, 2017, Nha Trang City, Viet Nam.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3155133.3155199

1 INTRODUCTION

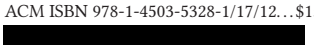
In order to minimize programmers' difficulties when developing parallel applications, a parallel programming tool at a higher level should be as easy-to-use as possible. MPI [11], which stands for Message Passing Interface, and OpenMP [12] are two widely-used tools that meet this requirement. MPI is 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, especially 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 [10], SCASH [15], the use of the RC model [9], performing a source-to-source translation to a tool like MPI [1, 3] or Global Array [8], or Cluster OpenMP [7].

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 [10] 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 that maps only the shared variables of the processes onto a shared-memory area attached to each process, the other variables being stored in a private memory, and the RC model that 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 an 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

SoICT '17, December 7–8, 2017, Nha Trang City, Viet Nam

ACM ISBN 978-1-4503-5328-1/17/12…$15.00



in the OpenMP standard). Thus, this one cannot be considered as a fully-compliant implementation of the OpenMP standard either.

In order to bypass these limitations, we developed CAPE [6, 14] 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) [5, 13] 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 [4, 17]. 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 an 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 [4, 6].

This paper presents a new execution model that improves the performance and the flexibility of CAPE. The paper is organized as follows: the next section describes in details CAPE mechanisms, capabilities and restrictions. Then Section 3 presents the design and the implementation of the new execution model together with its analysis in Section 4. Section 5 shows the result of the experimental analysis. Finally, Section 6 concludes the paper and presents our future works to improve CAPE.

2 CAPE PRINCIPLES

In order to execute OpenMP programs on distributed-memory systems, CAPE uses a set of templates to translate OpenMP source code into 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.

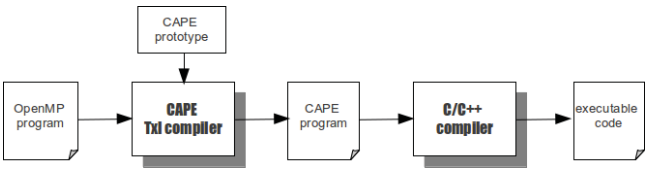


Figure 1: Translation OpenMP programs with CAPE.

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 is running 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) [4, 5] 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 makes it executable in the CAPE framework. So far, nested loops and shared-data variables constructs have not supported yet. However, this is not regarded as an issue as this can be solved at the level of the source-to-source translation and does not require any modification in the CAPE philosophy. In this end, CAPE can only be applied to OpenMP programs matching the Bernstein's conditions [2].

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 [3]. 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 name `file`.
- `inject(file)` injects a checkpoint into the memory of the current process.
- `send(file, node)` sends a checkpoint from current process to another.
- `wait_for(file)` waits for checkpoints and merges them to create another one.
- `merge(file`$_1$`,file`$_2$`)` merges two checkpoints together.
- `broadcast(file)` sends a checkpoint to all slave nodes.
- `receive(file)` waits for and receives a checkpoint.

2.3 Remarks

The good performance of CAPE as compared to those of MPI and the full compliance to the OpenMP specifications [4, 6] have made

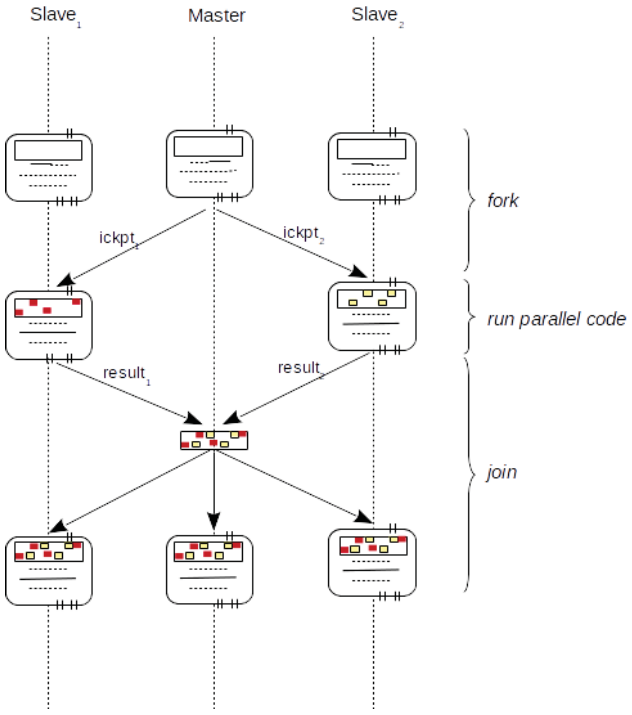


Figure 2: CAPE execution model.

CAPE a good alternative to port OpenMP on distributed-memory architectures. So far, the implementation of CAPE has not completed and will be improved in the following directions:

(1) As shown in the Figure 2, the master node might act as a bottleneck when waiting for checkpoints from the slaves, merging checkpoints and/or sending back data to slaves for memory synchronization.
(2) 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 (Figure 5). This method can reach a high-level of optimization. However, it might not be enough flexible for some cases as 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 can not be implemented using this method.
(3) After distributing the jobs, the slave nodes execute the divided jobs while the master does not do anything until the reception of the resulting checkpoints from the slaves, which clearly wastes resources.
(4) Nowadays, on most clusters, processors are equipped with more than two cores so that the performance can be significantly improved when using a multi-core architecture of each node in the system.
(5) OpenMP shared-data variable environment is an important element of OpenMP that needs to be supported by CAPE.

```
# 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.

3 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 Sec. 2.1. Figure 4 illustrates the model that 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 used to resume its execution in another process at the saved time.

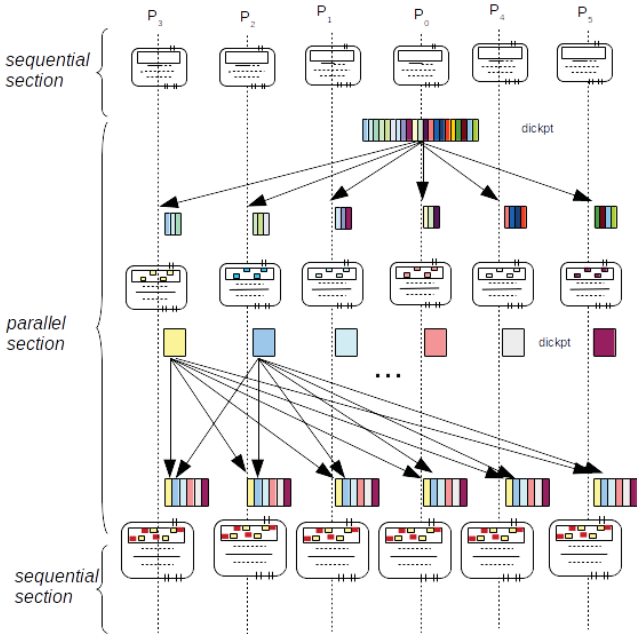


Figure 4: A new execution model for CAPE.

(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 6.
(4) The received incremental checkpoints are injected into the slave processes' memories.
(5) The slave processes resume their executions.
(6) Results on slave processes are extracted by identifying the modified regions and recorded into 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 it 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 7 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, the CAPE functions are as follows:

- $generate\_dickpt(before_i)$ (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.

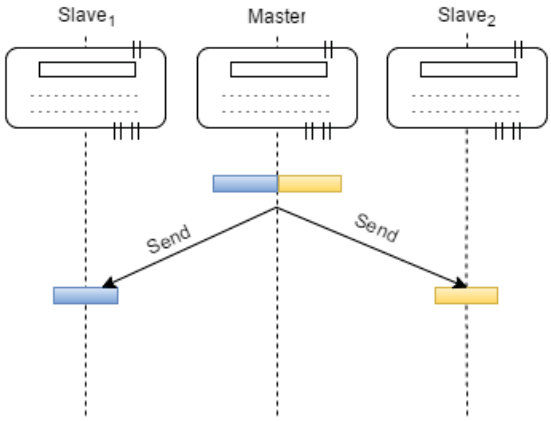


Figure 5: Scheduling method in CAPE-2.

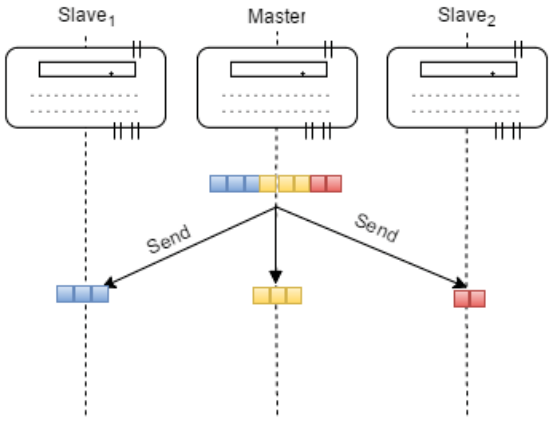


Figure 6: Scheduling method in new execution model.

- $allreduce(after_n, \&after, [< ops >])$ (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. This is performed using the Recursive Doubling algorithm [16] as illustration in Figure 8.
- $inject(after)$ (line 9): incremental checkpoint $after$ is injected into the application's memory to synchronize the state of the program on all nodes.

4 PERFORMANCE ANALYSIS AND EVALUATION

Moving from a scheduling of CAPE processes based on the number of nodes (Figure 5) to a scheduling based on the number of jobs (Figure 6) 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 number of jobs. First step, one checkpoint can be sent

```
# 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 7: Prototype for the *parallel for* for new model.

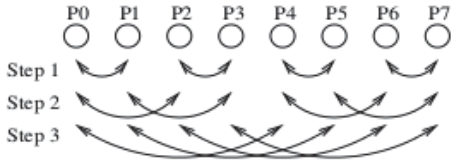


Figure 8: Recursive doubling for allreduce.

to each slave node to execute a divided job. When the slave node finishes, it will be 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.

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 incremental checkpoints only consist of very few bytes, ie. the fork phase time is close to zero.

For t_c, in the previous execution model, the master process was not involved in the calculation phase, it wastes the resources of the system. Assume that, we have n jobs and p processes, each process will take one unit of time to execute a job. And assume that, the number of jobs is divided equally among the processes. As a result, the value for t_c was:

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

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 one, the value for t_j was 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 was done sequentially. Thus, the time to send or receive a checkpoint can be given by:

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

With the new execution model, the Recursive Doubling algorithm [16] 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 is used more efficiently.

Besides, using 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.

5 EXPERIMENTS

In order to measure the impact of the new execution model on the performance, as mathematically analyzed in Section 4, 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 measurement campaign.

The program used as the basis for these experiments is the classic matrix-matrix multiplication. The sizes of the matrix are increased from 1600×1600, 3200×3200 to 9600×9600. Each program will be 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 9 and Figure 10 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) is represented for the previous execution model, and the blue one (NEW) is represented for the new execution model.

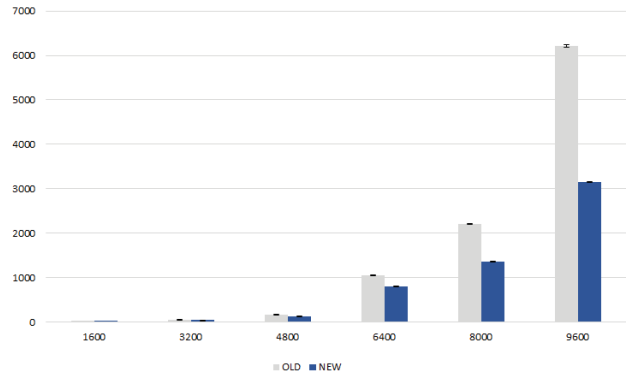


Figure 9: Total execution time (in seconds) of two models on 4-node clusters.

The horizontal axis shows the size of the matrix and vertical axis shows the execution time in seconds.

For 4-node clusters, 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 matrix is, the less time it takes. It is easy to understand that because there are only three nodes executed the divided jobs on the previous execution model. The master just divides and distributes jobs to slave, and then waits for the returned results. It does not participate in computational works. However,the new execution model is in contrast, after divided jobs, master node receives and executes a part of divided jobs. Therefore the calculation time (t_c) on this model is much lower than itself on previous model, especially, on the cluster which only has 4 nodes.

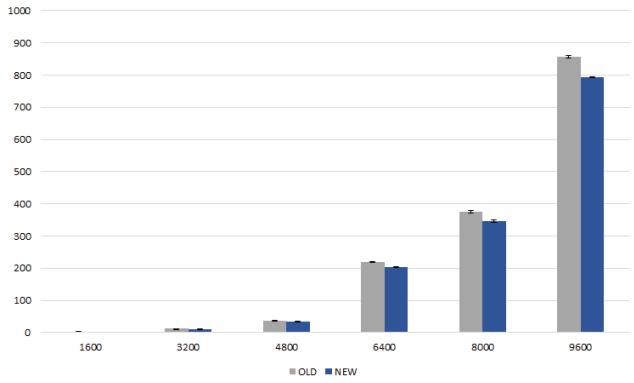


Figure 10: Total execution time (in seconds) of two models on 16-node clusters.

For 16-node cluster, the result in the Figure 10 shows the same trend, the new execution model is better than the previous one. However, the distance of execution time of two models is closer. It maintains a saving time around 10%. It is because the larger the number of nodes is, the less time it takes to calculate the divided jobs. Therefore, the total saving time of t_c in this case is not too much.

Figure 11 presents the execution time of the fork (t_f), the calculation (t_c) and the join (t_j) phases for both previous and new execution models on the master node on 16-node clusters. The matrix size 9600×9600 is selected in this case.

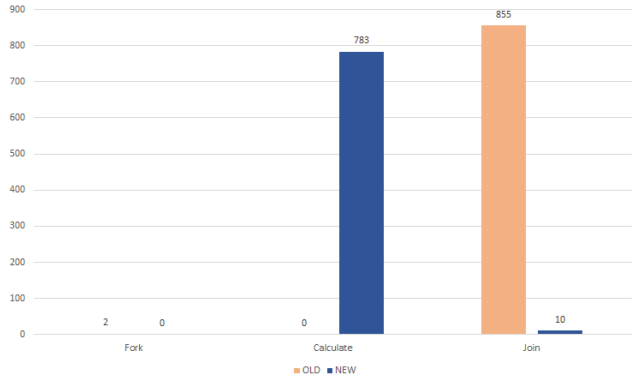


Figure 11: Execution time (in seconds) the fork, the calculation and the join phases for both previous and new execution models on the master node.

In the previous model, after the fork operation, the master node waits for the result 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 participate 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 to be compared 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.

6 CONCLUSION AND FUTURE WORKS

In this paper, we analysed the disadvantages of the previous execution model and proposed a new one to solve some of them. The theoretical analysis and experimentations showed that the execution time and the risk of bottlenecks are significantly reduced. Besides, a new scheduling method has been developed and presented to improve the flexibility of CAPE.

For the future works, we will keep on developing CAPE using this new execution model. We will use the operations on checkpoints to implement OpenMP's shared-data environment variables.

REFERENCES

[1] Ayon Basumallik and Rudolf Eigenmann. 2005. Towards automatic translation of OpenMP to MPI. In *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 189–198.

[2] Bernstein. 1966. Program Analysis for Parallel Processing. *IEEE Transaction on Electronic Computers* EC-15 (1966), 757–762.

[3] Antonio J Dorta, José M Badía, Enrique S Quintana, and Francisco de Sande. 2005. Implementing OpenMP for clusters on top of MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 148–155.

[4] Viet Hai Ha and Eric Renault. 2011. Design and performance analysis of CAPE based on discontinuous incremental checkpoints. In *2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*.

[5] Viet Hai Ha and Éric Renault. 2011. Discontinuous Incremental: A new approach towards extremely lightweight checkpoints. In *Computer Networks and Distributed Systems (CNDS), 2011 International Symposium on*. IEEE, 227–232.

[6] Viet Hai Ha and Eric Renault. 2011. Improving performance of CAPE using discontinuous incremental checkpointing. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE, 802–807.

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

[8] Lei Huang, Barbara Chapman, and Zhenying Liu. 2005. Towards a more efficient implementation of OpenMP for clusters via translation to global arrays. *Parallel Comput.* 31, 10 (2005), 1114–1139.

[9] Sven Karlsson, Sung-Woo Lee, and Mats Brorsson. 2002. A fully compliant OpenMP implementation on software distributed shared memory. In *High Performance Computing HiPC 2002*. Springer, 195–206.

[10] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, and Louis Rilling. 2003. Kerrighed: a single system image cluster operating system for high performance computing. In *Euro-Par 2003 Parallel Processing*. Springer, 1291–1294.

[11] MPI Forum. [n. d.]. Message Passing Interface Forum. Available at http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (2015-06-04). ([n. d.]).

[12] OpenMP ARB. 2013. OpenMP application program interface version 4.0. (2013).

[13] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1994. *Libckpt: Transparent checkpointing under unix*. Computer Science Department.

[14] Éric Renault. 2007. Distributed Implementation of OpenMP Based on Checkpointing Aided Parallel Execution. In *A Practical Programming Model for the Multi-Core Era*. Springer, 195–206.

[15] Mitsuhisa Sato, Hiroshi Harada, Atsushi Hasegawa, and Yutaka Ishikawa. 2001. Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *Scientific Programming* 9, 2, 3 (2001), 123–130.

[16] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[17] Van Long Tran, Eric Renault, and Viet Hai Ha. 2016. Analysis and evaluation of the performance of CAPE. In *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 International Symposium on*. IEEE, 620–627.