Analysis and evaluation of the performance of CAPE

van Long Tran, Eric Renault, Viet Hai Ha

Analysis and evaluation of the performance of CAPE

Van Long Tran and Eric Renault
SAMOVA, Telecom SudParis,
CNRS, Universite Paris-Saclay
9 rue Charles Fourier - 91011 Evry Cedex, France
Email: van_long.tran@telecom-sudparis.eu
and eric.renault@telecom-sudparis.eu

Viet Hai Ha
College of Education
Hue University
Hue, Vietnam
Email: haviethai@gmail.com

***Abstract*—MPI (Message Passing Interface) and OpenMP are two tools broadly used to develop parallel programs. On the one hand, MPI has the advantage of high performance while being difficult to use. On the other hand, OpenMP is very easy to use but is restricted to shared-memory architectures. CAPE is an approach based on checkpoints to allow the execution of OpenMP programs on distributed-memory architectures. This paper aims at presenting both an in-depth analysis and an evaluation of the performance of CAPE by comparing the execution model of both CAPE and MPI. Some suggestions are also provided to improve the use of CAPE.**

***Index Terms*—CAPE, OpenMP, MPI, high performance computing, parallel programming.**

I. 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, especially for non programmers, OpenMP is very easy to use, requesting the programmer to tag the pieces of the code to execute in parallel.

Some efforts have been made to port OpenMP on distributed-memory architectures. However, no solution has successfully met both 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, using SSI (for Single System Image) is the most straightforward approach. An SSI includes a DSM (for Distributed Shared Memory) 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 the SSI. However, as the shared memory is accessed through the network, the synchronization between the memories involves an important delay access time which makes this approach hardly scalable. Some experiments [3] show 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, like 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 to 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. Cluster OpenMP, proposed by Intel, also requires the use of additional indicators of its own (ie. not included in the OpenMP standard). As a result, this one cannot be considered as a fully-compliant implementation of the OpenMP standard.

To execute programs including OpenMP directives on distributed-memory architectures, CAPE (which stands for Checkpointing-Aided Parallel Execution) adopts a completely different approach based on the use of checkpointing techniques. This allows to distribute the work of the parallel block programs to the different processes of the system, and to handle the exchange of shared data automatically. Two versions of CAPE [10][11] have been released that proves the effectiveness of the approach and the high-performance.

This paper focuses on the in-depth analysis of the CAPE architecture and an evaluation of its performance. A comparison with MPI is proposed and some remarks are also provided for a better use of CAPE.

The next section presents an overview of some related work and serves as an introduction to MPI, OpenMP and checkpointing techniques. Section III introduces CAPE using the discontinuous incremental checkpointer. Section IV compares the operational model of both CAPE and MPI. Section V

presents some experimental results to support the analysis in Sec. IV. The last section outlines some conclusions and directions for future research.

II. Related Work

A. MPI

MPI [1] is an Application Programming Interface (API) developed during the 90s. This interface provides essential point-to-point communications, collective operations, synchronization, virtual topologies, and other communication facilities for a set of processes in a language-independent way, with a language-specific syntax, plus a small set of language-specific features. MPI uses the SPMD (Single Program Multiple Data) programming model where a single program is run independently on each node. A starter program is used to launch all processes, with one or more processes on each node and all processes first synchronize before executing any instructions. At the end of the execution, before exiting, processes all synchronize again. Any exchange of information can occur in between these two synchronizations. A typical example is the master-slave paradigm where the master distributes the computations among the slaves and each slave returns its result to master after the job complete.

Although it is capable of providing high performance and running on both shared- and distributed-memory architectures, MPI programs require programmers to explicitly divide the program into blocks, one for the master process and the others for slave processes. Moreover, some tasks, like sending and receiving data or the synchronization of processes, must be explicitly specified in the program. This makes it difficult for programmers for two main reasons. First, it requires the programmer to organize the program into parallel structures which are bit more complex structures than in sequential programs. This is even more difficult for the parallelization of a sequential program, as it severely disrupt the original structure of the program. Second, some MPI functions are difficult to understand and use. For these reasons, and despite the fact that MPI achieves high performance and provides a good abstraction of communications, MPI is still considered as assembly for parallel programming.

B. OpenMP

OpenMP [2] provides a higher level of abstraction than MPI for the development of parallel programs. It consists in a set of directives, functions and environment variables to easily support the transformation of a C, C++ or Fortran sequential program into a parallel program. A programmer can start writing a program on the basis of one of the supported languages, compile it, test it, and then gradually introduce parallelism by the mean of OpenMP directives. For C, C++ or Fortran compilers supporting OpenMP, directives are taken into consideration through the execution of specific codes. OpenMP offers directives for the parallelization of most classical parallel operations including loops, parallel sections, etc.

OpenMP follows the fork-join execution model with threads. When the program starts, it runs only one thread (the master thread) and executes the code sequentially. When it reaches an OpenMP parallel directive, the master thread spawns a team work including itself and a set of secondary threads (the fork phase), and the work allocated to each thread in this team starts. After secondary threads complete their allocated work, results are updated in the memory space of the master thread and all secondary threads are suspended (the join phase). Thus, after the join phase, the program executes only one thread as per the original. Fork-join phases can be carried out several times in a program and can be nested.

OpenMP is based on the use of threads. Thus, it can only run on shared-memory architectures as all threads use the same memory area. However, in order to speed up the computations, OpenMP uses the relaxed-consistency memory model. With this memory model, threads can use a local memory to improve memory accesses. The synchronization of the memory space of the threads is performed automatically both at the beginning and at the end of each parallel construct, or can be performed manually by specifying flush directives.

C. Checkpointing

Checkpointing aims at saving the state of a running program so that its execution can be resumed from that point later in time [12]. Checkpoints are used in many applications, for example to enable fault-tolerance or for backing up systems.

Checkpointing techniques can be divided into two groups. *Complete checkpointing* is a technique for which all the memory space and the associated information of a process is saved in each checkpoint. Its major drawbacks are the redundancy of data when several checkpoints are taken consecutively without many changes in the process memory space, and the large size of the checkpoints. *Incremental checkpointing* aims at saving only the part of the memory space and the associated information that have been updated since the initialization of the program or the last checkpoint. In order to identify the updated memory area, a checkpointed program is usually monitored by another one that performs the saving regularly.

CAPE requires taking checkpoints of each part of the continuing process. As a result, in order to serve CAPE in an optimal way, we developed a checkpointing technique called DICKPT [13] (which stands for Discontinuous Incremental Checkpointing). DICKPT is based on incremental checkpoints with additional image capabilities for discrete segments of the process. The analysis of experimental data has demonstrated the superior performance of this new capability both in terms of the reduction of the size of checkpoints and the reduction of the time to generate them.

III. CAPE base on Discontinuous Incremental Checkpointing

A. Execution model

CAPE is the alternative approach to allow the execution of OpenMP programs on distributed-memory systems. CAPE is based on a process as a parallel unit, which is different

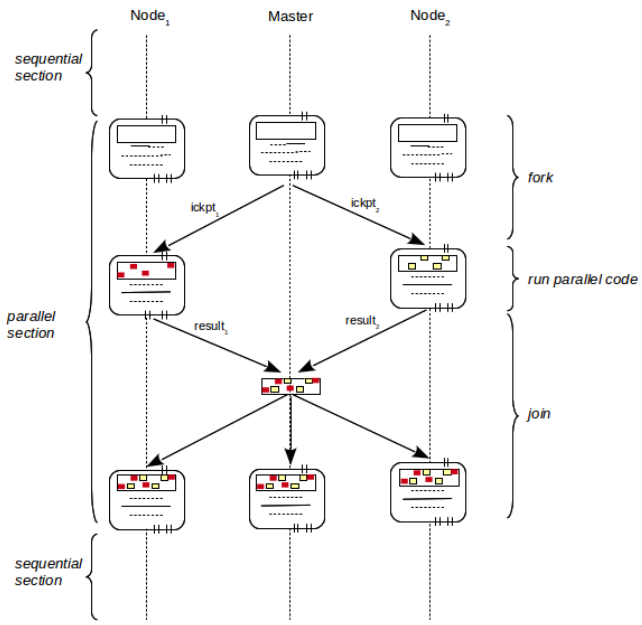

Fig. 1. The execution model of CAPE.

from the traditional implementations of OpenMP where the parallel unit is a thread. All the important tasks of the fork-join model are automatically generated by CAPE based on checkpointing techniques, such as task division, reception of results, updating results into the main process, etc. In its first version, CAPE used complete checkpoints so as to prove the concept. However, as the size of complete checkpoints is very large, it takes a lot of traffic on the network to transfer data between processes and involves a high cost for the comparison of the data from the different complete checkpoints to extract the modifications. These factors have significantly reduced the performance and the scalability of our solution. Fortunately, these drawbacks have been overcome in the second version of CAPE based on DICKPT.

Figure 1 describes the execution model of the second version of CAPE using three nodes. At the beginning, the program is initialized on all nodes and the same sequential code block is executed on all nodes. When reaching an OpenMP parallel structure, the master process divides the tasks into several parts and sends them to slave processes using DICKPT. Note that these checkpoints are very small in size, typically very few bytes, as they only contain the results of some very simple instructions to make the difference between the threads, which do not change the memory space that much. At each slave node, after receiving a checkpoint, it is injected into the local memory space and initialized for resuming. Then, the slave process executes the assigned task, extracts the result, and creates a resulting checkpoint. This last checkpoint is sent back to the master process. The master process then combines them altogether, injects the result into its memory space, sends it to all the other slave processes to synchronize the memory space of all processes and prepares for the execution of the next instruction of the program.

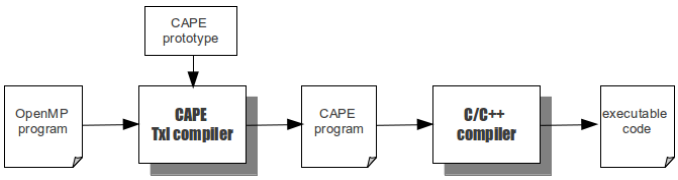

Fig. 2. Translation OpenMP programs with CAPE

B. Source-to-source translation template for the `parallel for` *construct*

In order to translate OpenMP programs, CAPE uses a set of templates. After the source-to-source translation of the OpenMP program, the code is regularly compiled by a C/C++ compiler. Figure 2 shows the different steps of the CAPE compilation process for a C/C++ OpenMP program. The current version of CAPE assumes that the parallel statements satisfy the Bernstein's conditions, which mean that all components of the code can be executed independently, no input and output data are common, and there is no communication between nodes during the execution.

Among all OpenMP parallel constructs, `parallel for` is probably the most important one, as 1) this is the most used construct in user programs and 2) it can act as an intermediate structure to convert other OpenMP parallel constructs including for example the `parallel sections` one. As a result, instead of building a complete template for each construct, some are first translated into the `parallel for` construct. CAPE automatically translates the `parallel for` construct to a set of function calls that contains the fundamental operations of CAPE. The template for the `parallel for` construct is shown in Figure 3. In order to make the presentation easier to understand, the template is presented in a version where the number of iterations of the `for` statement is equal to the number of threads. Below is the list of basic functions used by CAPE.

1) `start ( )` set the environment to create DICKPT.
2) `stop ( )` suspends the environment to create checkpoints.
3) `create ( file )` create a checkpoint and save it to *file*.
4) `inject ( file )` updates the current process with the information contained in checkpoint *file* provided as a parameter.
5) `send ( file, node )` sends the content of *file* to *node*.
6) `wait_for ( file )` waits and merges all the components of the *file* .
7) `last_parallel ( )` returns `TRUE` when the current parallel block is the last one of the entire program and `FALSE` otherwise.
8) `merge (` $file_1$ `,` $file_2$ `)` applies the list of modifications from $file_2$ to checkpoint $file_1$.
9) `broadcast ( file )` sends *file* to all slave nodes.
10) `receive ( file )` waits for *file* to be available.

The main steps of CAPE can be explained as follows:

On the master side (lines 1 to 12), the first statement of the for loop is executed. At each `for` loop iteration, the master distributes jobs to each slave by generating and sending it as an incremental checkpoint. Then, it waits for the results from the slave process. After the result is received, it updates the memory. Finally, the master node broadcasts the result to all slave nodes to guarantee the memory consistency if necessary.

On the slave process side (lines 13 to 25), each process receives a DICKPT from the master. Then, the checkpoint is injected into the memory process space which executes the assigned tasks. At last, the result of the execution on the slave node is extracted into an incremental checkpoint and sent back to the master node to update its memory.

```
# 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
```

Fig. 3. Template for the *parallel for* with incremental checkpoints.

IV. Analysis and performance evaluation

A. MPI and CAPE are almost the same...

Similarities between CAPE and MPI are many including the architectural background, organizational models, and execution models. From the architectural point of view, MPI and CAPE both have the ability to execute on multi-processor architectures, and the most important goal for both is to support systems based on a distributed memory such as clusters, grids, and clouds. From the organizational- and programming-model point of view, both use processes and the master-slave paradigm, with a single process attached to a node. Execution models for CAPE and MPI contain similar steps. To execute a program that contains parallel and sequential code alternatively, the steps are i) the initialization of the program, and then ii) the execution of both sequential and parallel sections on a set of processes. The initialization is the same for CAPE and MPI. All nodes are initialized at the beginning of the execution and one of the nodes is chosen as the master, the others being considered as slaves.

B. ... but not exactly

The main difference between CAPE and MPI is the determination of the data to exchange and the way data in the memory space of a process are updated. MPI was defined in the scope of a third generation language. The user is required to explicitly define what (the address, the length and the data type) and when (send and receive) to transfer the memory areas. In order to do so, MPI is based on a set of functions which requires the user to specify the operations that have to be executed step by step. However, CAPE is based on OpenMP which was defined as a fourth generation language. In this case, the user specifies the expected result and it is the compiler work to automatically translate this objective into effective code. For the specific case of CAPE, everything is automatically done through the use of checkpoints. The result of the execution of a code block is saved in a discontinuous incremental checkpoint initialized before executing the block and saved at the end of the block. The checkpoint is injected into the memory space of the target process using the `inject` function (as shown above).

The ability for CAPE to automatically distribute the work, to extract and inject the results into the memory space of the process, creates an advantage in the transparency of the communication of data between processes for the programmer. This definitively helps the implementation of a fully-compatible version of OpenMP for distributed-memory architectures. At the same time, the use of checkpoints is also the main drawback of CAPE as it slows down the execution.

C. Qualitative comparison

Table I presents a qualitative comparison of the different steps when executing a parallel code with both MPI and CAPE. As mentioned above, steps for initialization and execution are similar for both CAPE and MPI. As a result, they are not mentioned on the table.

From Table I, let t_b be the time for work division and distribution to the slaves, t_c be the computation time, t_u be the time for result extraction and returning the data to the master, and t_s be the time for the memory update on all processes.

TABLE I
THE STEPS THAT EXECUTE A PARALLEL CODE BLOCK OF CAPE AND MPI

Steps	CAPE		MPI	
	Approach	Pros and cons	Approach	Pros and cons
Work division and distribution to slaves. (t_b)	The master automatically divides the task using DICKPT and sends them to the slaves.	Transparent to the programmer. Not much time wasted on running.	The programmer divides the code into tasks for all processes.	Not straightforward.
Computation. (t_c)	Slaves execute the application code under the supervision of DICKPT.	Overhead due to the monitoring of the process by the checkpointer.	Processes execute the application code independently.	High performance
Result extraction and return to the master. (t_u)	Results automatically extracted and sent back using DICKPT.	Transparent for the programmer. Accurate extraction of results. Overhead spent on filtering results.	The programmer writes the code to determine data to send and where to store the results.	High performance. Difficult for the programmer.
Memory update on all processes. (t_s)	Memory space of all processes automatically updated by DICKPT.	Transparent for the programmer. Automatic synchronization. Faster than MPI.	The programmer writes the synchronization code.	Difficult for the programmer.

For any given program, let t_{tt} and t_{ss} be the execution time of the sequential part and the parallel part of the code respectively. Let t be the total amount of time to execute the program. It is given by:

$$t = \sum t_{tt} + \sum t_{ss} \quad (1)$$

$$= \sum t_{tt} + \sum (t_b + t_c + t_u + t_s) \quad (2)$$

Note that executing a program in parallel is meaningful when the execution time of the parallel part of a program is far greater than the execution time for the initialization and the communication between the nodes. It is the same for the time spent in the synchronization of the memory space of the processes. As a result, in the following, the time for initialization and communication as negligible as compared to the time for computation and synchronization, ie. $t_b = t_s = 0$. This leads to:

$$t = \sum t_{tt} + \sum (t_c + t_u) \quad (3)$$

Then, it is fair to consider that the sequential code executed on all the nodes with both MPI and CAPE is the same and thus that the time is equal for both techniques. The time difference between CAPE and MPI becomes:

$$\Delta t = \sum \Delta (t_c + t_u) \quad (4)$$

Basically, the computation of the parallel code should be the same for MPI and CAPE, and so the time difference in t_c as the cost of monitoring and generating DICKPT, in the application process is always monitored by a checkpointer process. Therefore, the execution time of CAPE applications is higher than the one of MPI. According to [14], the time difference is in the 2% to 20%. The time difference in t_u caused by the process update via DICKPT is more complex. However, for the problems have the high t_c, so the t_u is too small in the total time of the program. Therefore we can ignore that time.

TABLE II
EXECUTION TIME (IN SECONDS) ON A SINGLE NODE.

Size	Sequential	OpenMP
3000 x 3000	258.9	142.4
6000 x 6000	1852.7	1048.7
9000 x 9000	7314.5	3986.2
12000 x 12000	14990.5	8999.4

V. EXPERIMENTAL RESULTS

In order to validate our approach, some performance measurements have been conducted on a Desktop Cluster. This testbed is composed of nodes including Intel(R) Core(TM)2 Duo E8400 CPUs running at 3 GHz and 2 GB RAM, operated by Linux kernel 2.6.35 with the Ubuntu 10.10 flavor, and connected by a standard Ethernet at 100 Mb/s. In order to avoid as much as possible external influences, the entire system was dedicated to the tests during performance measurements.

The program used for tests is a matrix-matrix product for which the size varies from 3,000×3,000 to 12,000×12,000. Matrices are supposed to be dense and no specific algorithm has been implemented to take into account sparse matrices. Each experiment has been performed at least 10 times and a confidence interval of at least 90% has always been achieved for the measures. Data reported here are the means of the 10 measures.

The execution of both the sequential version and the OpenMP version of the program on one of the nodes gives the result provided in Table II. A single core was used for the sequential execution of the program, while the OpenMP program takes benefits of the two cores. One can check that results in Table II are consistent as the execution time for both sequential and OpenMP versions are directly proportional to the cube of the matrix size. Typically, this means that no important cache effects have polluted the performance measurements, probably because almost all data fit into memory. Moreover, the speedup obtained by OpenMP is 1.8 for the first three matrix sizes and 1.65 for the last one, which are expected values.

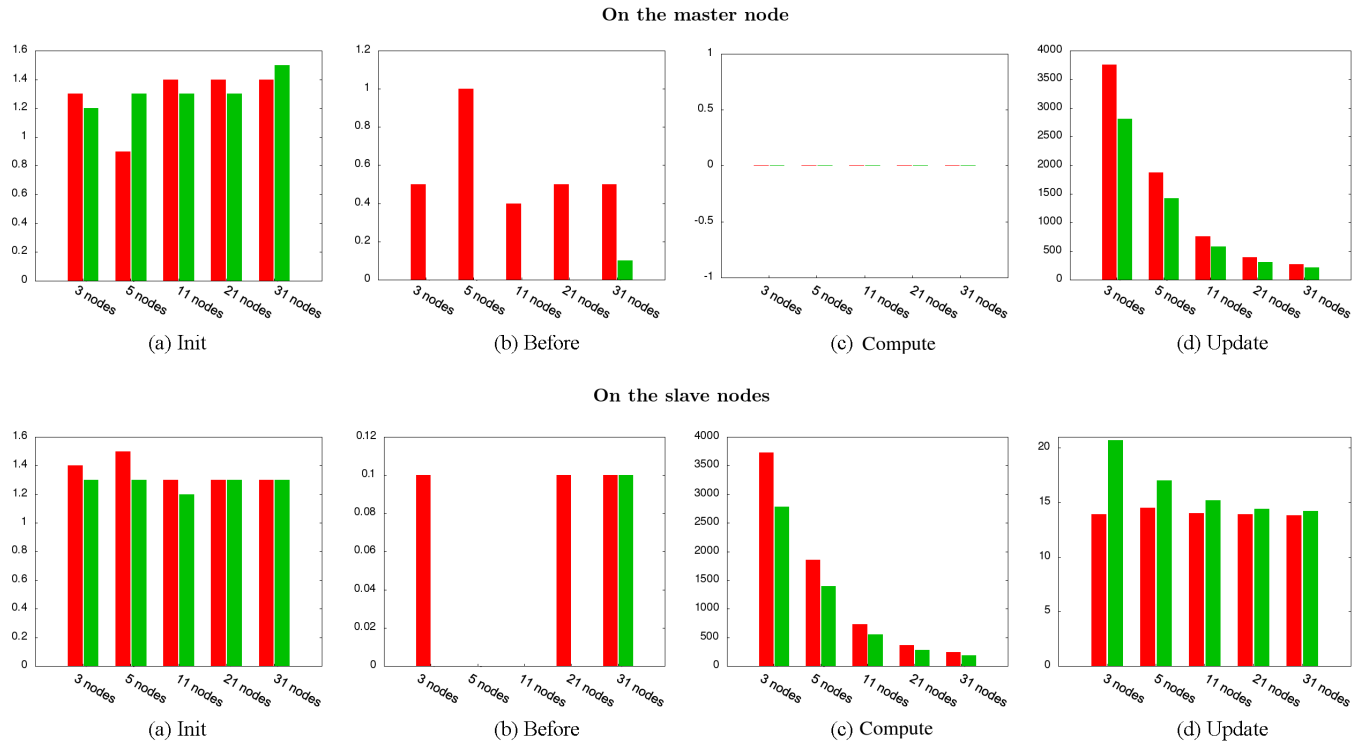

Fig. 4. Execution time (in seconds) vs. number of nodes.

The next part of the experiment consists in comparing CAPE and MPI. Figure 4 and 5 present the execution time in seconds of the matrix-matrix program for various numbers of nodes and matrix sizes. Note that, despite the fact that processors are dual core, a single core was used during the experiments. Two measures are represented each time: the left one is associated with CAPE using incremental checkpoints, and the right one is associated with MPI. Both Figure 4 and 5 contain two groups of graphs. The group above belongs to the master node, while the one below belongs to the slaves. The line numbers for the code refer to Figure 3. Each group shows four graphs and the meaning is as follows:

- *Init* is the elapsed time between the beginning of the program and the beginning of the parallel for loop in the matrix-matrix product. On Fig. 3, these are all lines before the first one. In the modeling above, this is time t_{tt}.
- *Before* is the time spent to create and send checkpoints (lines 2 to 5) on the master node. On slave nodes, this includes waiting for and receiving the checkpoint, and then updating the slave process using the checkpoint (lines 16 and 17). For the case of MPI, this is the time to send data to slave nodes. In the modeling above, this is time t_b.
- *Compute* is the time to generate the last checkpoint on the master node (lines 7 and 8) and the time to do the job (execute the code of matrix-matrix product) on the slaves (lines 18 and 19). In the modeling above, this is time t_c.
- *Update* is the time spend in waiting for and receiving all updates from the slave nodes and inject them into the master process memory (lines 9 and 10). On slave nodes, this is the time to generate the incremental checkpoints and send them to the master node (lines 20 to 22). For the case of MPI, this is the time to send results from slave nodes to the master node. In the modelling above, this is time t_u.

Figure 4 presents the execution time in the detailed phases with different numbers of nodes. The size of matrices are 9,000×9,000. However, similar trends are observed for the other matrix sizes. In the master node, the execution times for the `Init`, `Before` and `Compute` steps are small as compared to step `Update` (t_{tt}, t_b and t_c are very small as compared to t_u). This is because step `Update` waits for the results computed in the slaves. This means that t_u includes the computation time of the slaves to do the assigned tasks. Then, one can see that t is similar to t_u on the master node. Similarly to the slaves, the execution times of both `Before` and `Init` steps are too small and can be ignored. As a result, the total execution time consists in both the computation time and the update time. This clearly conforms to equations 3 and 4.

Graphs in Figure 4 also show that, for the case with three nodes, the execution time for CAPE is always larger than the one for MPI. This is due to the fact that slaves compute and update the result of half of the matrix, and the size of the checkpoint takes a large part of the total memory space

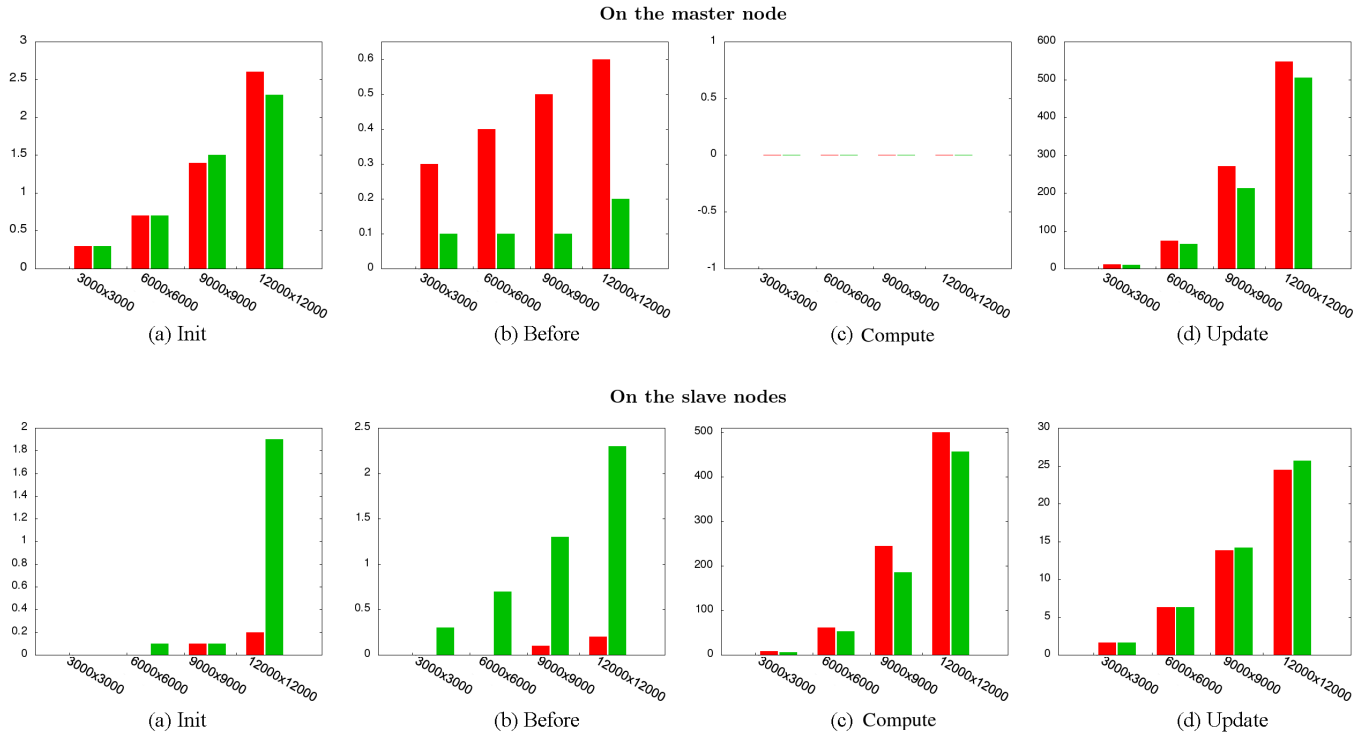

Fig. 5. Execution time (in seconds) vs. problem size.

of the process. Therefore, the impact of the checkpointer on the total execution time is very important. However, when increasing the number of machines, the size the checkpoints decreases together with Δt. When increasing the number of nodes to 31, this time is approximately the same for both CAPE and MPI. This shows that the performance of CAPE is poor in systems with a very small number of nodes. However, no one would use a distributed-memory machine to run a program requiring a small number of threads. Any shared-memory machine would be sufficient in that case. On the other hand, the same graphs also show that CAPE has the ability to achieve high performance when the size of checkpoints is small. This is involved by the fact that slaves only update a small amount of memory space.

Figure 5 presents the execution time for different matrix sizes for a matrix-matrix product. Thirty one computers have been used to perform the parallel computations. However, the results tend to be similar to the experimental system when the numbers of nodes is different (except for the case with 3 nodes). Comments to chart in Figure 4 are also true for this one. The data on the graph shows that the execution time for both MPI and CAPE are proportional to the size of the matrix. As both MPI and CAPE are only using the data transfer mechanism to update the computation results, the execution speed is also quite similar. In a more detailed analysis, data show that the total execution time for CAPE based on DICKPT is only about 10% higher than the execution time for MPI, except for matrices with a size of 3000x3000 which rate is 1.3.

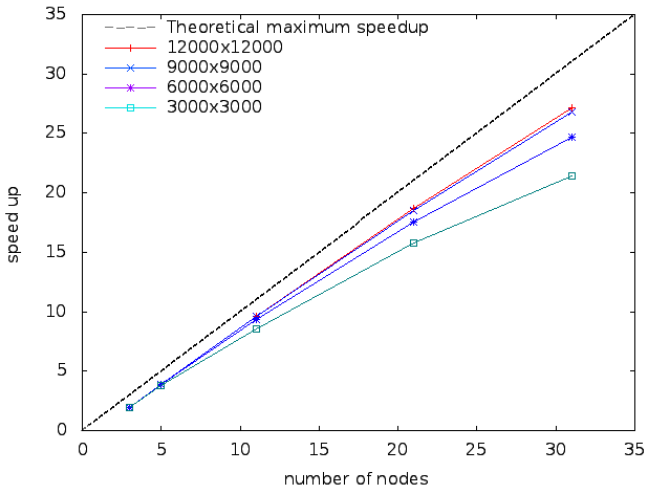

Fig. 6. Speedup vs. number of nodes.

Figure 6 shows the speedup of CAPE using incremental checkpoints for various number of nodes and matrix sizes. The dotted line represents the maximum theoretical speedup. The figure clearly shows that the solution provides an efficiency (the ratio of the speedup over the number of nodes) in the range from 75% to 90% which is very good. Also, it highlights that the larger the size of matrices, the higher the speedup.

VI. Conclusion and future work

CAPE, with its ability to automatically divide the task and establish data communication between processes, has demonstrated its ability to install a conforming implementation of OpenMP on distributed-memory systems. Moreover, the comparison of the performance results of MPI and CAPE by means of both a theoretical and an experimental analysis has proven that the efficiency of CAPE programs is very close to the one of MPI programs, and that the difference between both is getting smaller and smaller as the size of the problem and the number of nodes in the system increases.

For the near future, we will keep on developing CAPE to support the OpenMP constructs that have not been ported yet. Besides, we would like to study other models to take benefits of CAPE so that our model can improve the flexibility of the execution of many more sequential programs.

References

[1] "Message passing interface forum." [Online]. Available: http://www.mpi-forum.org/

[2] A. OpenMP, "Openmp application program interface version 4.0," 2013.

[3] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath, and L. Rilling, "Kerrighed: a single system image cluster operating system for high performance computing," in *Euro-Par 2003 Parallel Processing*. Springer, 2003, pp. 1291–1294.

[4] M. Sato, H. Harada, A. Hasegawa, and Y. Ishikawa, "Cluster-enabled openmp: An openmp compiler for the scash software distributed shared memory system," *Scientific Programming*, vol. 9, no. 2, 3, pp. 123–130, 2001.

[5] S. Karlsson, S.-W. Lee, and M. Brorsson, "A fully compliant openmp implementation on software distributed shared memory," in *High Performance ComputingHiPC 2002*. Springer, 2002, pp. 195–206.

[6] A. Basumallik and R. Eigenmann, "Towards automatic translation of openmp to mpi," in *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005, pp. 189–198.

[7] A. J. Dorta, J. M. Badía, E. S. Quintana, and F. de Sande, "Implementing openmp for clusters on top of mpi," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2005, pp. 148–155.

[8] L. Huang, B. Chapman, and Z. Liu, "Towards a more efficient implementation of openmp for clusters via translation to global arrays," *Parallel Computing*, vol. 31, no. 10, pp. 1114–1139, 2005.

[9] J. P. Hoeflinger, "Extending openmp to clusters," *White Paper, Intel Corporation*, 2006.

[10] É. Renault, "Distributed implementation of openmp based on checkpointing aided parallel execution," in *A Practical Programming Model for the Multi-Core Era*. Springer, 2007, pp. 195–206.

[11] V. H. Ha and E. Renault, "Improving performance of cape using discontinuous incremental checkpointing," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE, 2011, pp. 802–807.

[12] J. S. Plank, "An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance," Technical Report UTCS-97-372, Tech. Rep., 1997.

[13] V. H. Ha and É. Renault, "Discontinuous incremental: A new approach towards extremely lightweight checkpoints," in *Computer Networks and Distributed Systems (CNDS), 2011 International Symposium on*. IEEE, 2011, pp. 227–232.

[14] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, p. 9.