CAPE: A Checkpointing-Based Solution for OpenMP on Distributed-Memory Architectures

Van Long Tran[1,2(✉)], Éric Renault[2], and Viet Hai Ha[3]

[1] Hue Industrial College, 70 Nguyen Hue Street, Hue City, Vietnam
tvlong@hueic.edu.vn
[2] SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay, 9 rue Charles Fourier, 91011 Evry Cedex, France
eric.renault@telecom-sudparis.eu
[3] College of Education, Hue University, Hue, Vietnam
haviethai@gmail.com

Abstract. CAPE, which stands for Checkpointing-Aided Parallel Execution, is a framework that automatically translates and provides runtime functions to execute OpenMP programs on distributed-memory architectures based on checkpointing techniques. 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 which is then compiled using a regular C/C++ compiler. This code can be executed on distributed-memory systems under the support of the CAPE framework.

This paper aims at presenting the design and implementation of a new execution model based on Time-stamp Incremental Checkpoints. The new execution model allows CAPE to use resources efficiently, avoid the risk of bottlenecks, overcome the requirement of matching the Bernstein's conditions. As a result, these approaches make CAPE improving the performance, ability as well as reliability.

Keywords: CAPE · Checkpointing aided parallel execution · OpenMP on cluster · Parallel programming · Distributed computing · HPC

1 Introduction

OpenMP and MPI have become the standard tools to develop parallel programs on shared-memory and distributed-memory architectures respectively. As compared to MPI, OpenMP is easier to use. This is due to its ability to automatically execute code in parallel and synchronize results using its directives, clauses, and runtime functions while MPI requires programmers to do all this manually. Therefore, some efforts have been made to port OpenMP on distributed-memory architectures. However, excluding CAPE [7,9,18], no solution has successfully

© Springer Nature Switzerland AG 2019
V. Malyshkin (Ed.): PaCT 2019, LNCS 11657, pp. 93–106, 2019.
https://doi.org/10.1007/978-3-030-25636-4_8

met these two requirements: (1) to be fully compliant with the OpenMP standard and (2) high performance. Most prominent approaches include the use of an SSI [15], SCASH [19], the use of the RC model [13], performing a source-to-source translation to a tool like MPI [1,5] or Global Array [12], or Cluster OpenMP [11].

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

CAPE used the Discontinuous Incremental Checkpointing (DICKPT) [8] to implement the OpenMP fork-join model. The jobs of OpenMP work-sharing constructs are divided and distributed to slave nodes using checkpoints. At each slave node, these checkpoints are used to resume execution. In addition, the results after executing the divided jobs on each slave node are also extracted using checkpoints and sent back to the master. It has been demonstrated that this solution is fully compliant with OpenMP and provides high performance. However, there are some limitations:

- to run on top of CAPE, an OpenMP program must fulfill the Bernstein's conditions. This is the reason why the matrix-matrix product has been extensively used in the previous experiments.
- The implementation of CAPE wastes the resources. In the implementation of OpenMP work-sharing constructs on CAPE, the master does not perform a part of the computation. It waits for checkpoint results from the slave nodes and merges them together.

- The risk of bottlenecks and low communication performance at the implementation of the join phase. After executing the divided jobs, each slave node extracts a result checkpoint and sends it back to the master. The master receives, merges checkpoints together and sends the result back to the slave nodes in order to synchronize data.

This paper presents the design and implementation of a new model for CAPE based on Time-stamp Incremental Checkpointing (TICKPT) [24] to bypass the drawbacks mentioned above. The new implementation based on TICKPT improves the performance, capability, and reliability of this solution.

2 Checkpoint Techniques

2.1 Checkpointing

Checkpointing is the technique that saves the image of a process at a point during its lifetime, and allows it to be resumed from the saving's time if necessary [4,17]. Using checkpointing, processes can resume their execution from a checkpoint state when a failure occurs. So, there is no need to take time to initialize and execute it from the begin. These techniques have been introduced for more than two decades. Nowadays, they are used widely for fault-tolerance, applications trace/debugging, roll-back/animated playback, and process migration. To be able to save and resume the state of a process, the checkpoint saves all necessary information at the checkpoint's time. It can include register values, process's address space, open files/pipes/sockets status, current working directory, signal handlers, process identities, etc. The process's address space consists of text, data, *mmap* memory area, shared libraries, heap, and stack segments. Depending on the kind of checkpoints and its application, the checkpoint takes all or some of these information.

Based on the structure and contents of the checkpoint file, checkpointings are categorized into two groups: complete and incremental checkpointing.

- Complete checkpointing [3,4,14] saves all information regarding the process at the points that it generates checkpoints. The advantages of this technique are the reduction of the time of generation and restoration. However, not only a lot of duplicated data are stored each time a checkpoint is taken, there are also duplications in the different generated checkpoints.
- Incremental checkpointing [8,10,17] only saves the modified data. This has to be compared with the previous checkpoint. This technique reduces checkpoint's overhead and checkpoint's size. Therefore, it is widely used in distributed computing.

2.2 Time-Stamp Incremental Checkpointing

Time-stamp Incremental Checkpointing (TICKPT) [24] is an improvement of DICKPT by adding new factor – time-stamp – into incremental checkpoints and by removing unnecessary data based on data-sharing variable attributes of OpenMP programs.

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.

3 CAPE Based on TICKPT

3.1 Abstract Model

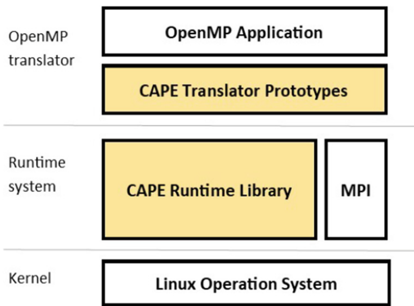

Fig. 1. New abstract model for CAPE.

Figure 1 presents the new abstract model for CAPE. It is designed based on TICKPT and uses MPI to transfer data over the network.

As presented in the previous version [21,22], CAPE provides a set of prototypes to translate OpenMP codes into CAPE codes. An OpenMP CAPE code in C or C++ is replaced by a set of calls to CAPE runtime functions. In this version, the CAPE translator prototypes are modified and added to adapt to the new mechanism based on TICKPT. This provides a set of prototypes to translate the common constructs, clauses, and runtime functions of OpenMP.

For the CAPE Runtime library, apart from providing functions to handle OpenMP instructions and to port them on distributed memory systems, some functions have been added to manage the declaration of variables and the allocation of memory on the heap. To transfer data among nodes in the system, instead of using the functions based on sockets like in the previous version, `MPI_Send` and `MPI_Recv` functions are called to ensure high reliability.

3.2 RC-Model Based CAPE Memory Model Implementation

OpenMP uses the Relaxed Consistency (RC) memory model. This model allows shared memory allocated in the local memory of a thread to improve memory accesses. When a synchronization point is reached, this local memory is updated in the shared memory area that can be assessed by all threads.

```
C_id ← generate_checkpoint(flag);
C ← all_reduce (C_id, id, nnodes,
      [operators]);
inject(C) ;
```

Fig. 2. `cape_flush()` implementation.

CAPE completely implements the RC model of OpenMP on distributed-memory systems. All variables, including private and shared variables, are stored at all nodes of the system, and they can be only accessed locally. At synchronization points, only the modified data of shared variables at each node are extracted and saved into a checkpoint. This checkpoint is sent to the other nodes in the system, and is merged using the `merging` checkpoint operation with the other. Then, the result checkpoint is injected into the application memory to synchronize data.

In the CAPE runtime library, there are two fundamental functions which are called implicitly at synchronization points:

- `cape_flush()` generates a TICKPT, gathers, merges, and injects them into the application memory. This function is described by pseudo code in Fig. 2. Here, the `all_reduce()` function is responsible for gathering and merging the checkpoints generated by the `generate_checkpoint()` function. The gathering and the merging is implemented using both Ring and Recursive Doubling algorithm. The algorithm is automatically selected to be executed by the system depending on the size of the checkpoint.
- `cape_barrier()` sets a barrier and updates shared data between nodes. This function calls `MPI_Barrier()` of the MPI runtime library, and then uses `cape_flush()` to update shared data.

3.3 Execution Model

Figure 3 illustrates the execution model of CAPE. The idea of this model is the use of TICKPT to identify and synchronize the modified data of shared variables of the program among the nodes. OpenMP threads are replaced by processes, and each process runs in a node. At the beginning, the program is initialized and executed at the same time in all nodes of the system. Then, the execution works as the following rules:

- The sequential region or the code inside the `parallel` construct but not belonging to any other constructs is executed in the same way for all nodes.
- When the program reaches a `parallel` region, on each node, CAPE detects and saves the properties of all shared variables that are implicitly declared as sharing. If there are any OpenMP clauses declared in the `parallel` construct,

the relevant runtime functions are called to modify variable properties. Then, the `start` directive of TICKPT is called to save the value of the shared variables.
– At the end of a `parallel` region, the implementation of the `barrier` construct is implicitly called to synchronize data, and the `stop` directive of TICKPT is called to remove all relevant data.
– For the loop construct, each node (including the master node) is responsible for computing a part of the work based on the re-calculation of the range of iterations.
– For the `sections` construct, each node is divided into one or more parts of works that are indicated using `section` construct.
– At the `barrier`, the implementation of the `flush` construct is called to synchronize data.
– When the program reaches the `flush` construct, a TICKPT is generated and synchronized among the nodes to update the modification of shared data. According to [16], a `flush` is implicit at the following locations:
  • At the `barrier`.
  • At the entry to and the exit from `parallel`, `critical`, and `atomic` constructs.
  • At the exit from `for`, `sections`, and `single` constructs unless a `nowait` clause is present.

In this execution model, instead of using the master node to divide jobs and distribute to slave nodes based on incremental checkpoints in order to implement OpenMP work-sharing constructs, each node computes and executes the divided jobs automatically. At synchronization points, a TICKPT is generated at each node. It contains the modified data of shared variables and their time-stamps after executing the divided jobs. These checkpoints are gathered and merged at all nodes in the system using the Ring or Recursive Doubling algorithm [20]. This allows CAPE to void the bottleneck and improve the performance of communication tasks.

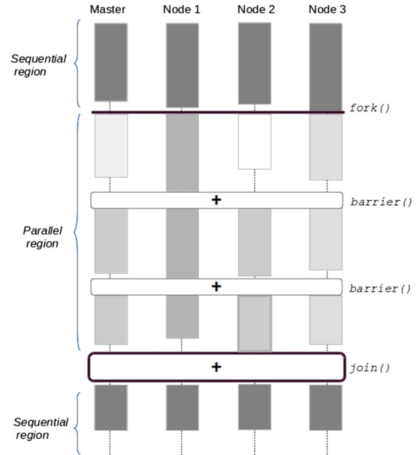

Fig. 3. The new execution model of CAPE.

With the features of TICKPT, checkpoints are able to use checkpoint's operations [23,24]. This allows memory elements to share the same address when computing and makes it simple when merging. Therefore, it allows CAPE to work without the need for the program to match with the Bernstein's conditions. Moreover, the master node takes a part in the computation of the divided jobs. This uses all the resources and improves the system efficiency.

The only missing part of the OpenMP specifications for this implementation is that `dynamic` and `guided` scheduling directives of the work-sharing construct

have not been implemented yet. However, one can demonstrate that they can be easily translated into a `static` scheduling.

3.4 Prototypes

To be executed on a distributed-memory system with the support of the CAPE runtime library, the OpenMP source code is translated into a CAPE source code. There, each construct, clause, and runtime function of the OpenMP source code is translated into the relevant runtime function of CAPE. This translation works under the provision of a set of CAPE prototypes.

Based on the general syntax of OpenMP directives, a general template for CAPE prototypes was designed and is illustrated in Fig. 4. They are as follows:

```
cape_begin(directive-name, param-1, param-2);
     [cape_clause_functions]
     ckpt_start();
     //code blocks
cape_end(directive-name, reduction-flag );
```

Fig. 4. General template for CAPE prototypes in `C/C++`.

- `cape_begin()` and `cape_end()` are CAPE runtime functions which perform the actions for entering and exiting OpenMP directives. The `directive-name` is a label declared by CAPE which corresponds to the relevant CAPE runtime function. Depending on this label, the `cape_barrier()` function is called to update the shared data of the system. `param-1` and `param-2` are used to store the range of iterations for `for` loops, otherwise they both are set to zero. The `reduction-flag` is set to `TRUE` if there is a declaration of OpenMP reduction clause, otherwise it is set to `FALSE`.
- `cape_clause_functions` is a set of CAPE runtime functions which is used to implement OpenMP clauses. This implementation is presented in [23].
- `ckpt_start()` marks the location where to start the checkpointing. When reaching the `ckpt_start()` function, the value of shared variables is copied.

4 Experiments

In order to evaluate the performance of this new approach, we designed a set of micro benchmarks and tested them on a Desktop Cluster. The designed programs are based on the Microbenchmark for OpenMP 2.0 [2,6]. These programs have been translated to CAPE and executed on a Cluster to compare the performance.

4.1 Benchmarks

(1) MAMULT2D: This program computes the multiplication of two matrices. Originally, it was written in C/C++ and used the OpenMP `parallel for` construct. It matches Bernstein's conditions. Therefore, it has been used extensively to test CAPE in the previous works.

```
int vector(float A[], float B[], float C[], float D[], int n){
        int i, nthreads, tid;
        #pragma omp parallel shared(C,D,nthreads) private(A, B, i,tid)
      {
                tid = omp_get_thread_num();
                if (tid == 0)
                {
                        nthreads = omp_get_num_threads();
                        printf("Number of threads = %d\n", nthreads);
                }
                printf("Thread %d starting...\n",tid);
                #pragma omp sections nowait
                {
                        #pragma omp section
                printf("Thread %d doing section 1\n",tid);
                        for (i=0; i<N; i++)
                        {
                                for (j= 0 ; j< N; j+=25)
                                        A[j] = A[j] * 0.15 ;
                                C[i] = A[i] + B[i];
                                printf("Thread %d: C[%d]= %f\n",tid,i,C[i]);
                        }
                        #pragma omp section
                        printf("Thread %d doing section 2\n",tid);
                        for (i=0; i<N; i++)
                        {
                                for (j= 0 ; j< N; j+=25)
                                        B[j] = B[j] + 10.25 ;
                                D[i] = A[i] * B[i];
                                printf("Thread %d: D[%d]= %f\n",tid,i,D[i]);
                        }
                } /* end of sections */
        } /* end of parallel section */
        return 0;
}
```

Fig. 5. OpenMP function to compute vectors using `sections` construct.

(2) PRIME: This program counts the number of prime numbers in the range from 1 to N. The OpenMP code uses the `parallel for` construct with data-sharing clauses.

(3) PI: This program computes the value of PI by mean of the numeric integration method using Eq. (1).

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \tag{1}$$

(4) VECTOR-1: This program performs operations on vectors. It contains OpenMP runtime functions, data-sharing clauses, a `nowait` clause, and `parallel` and `sections` constructs. The OpenMP code is presented in Fig. 5.

(5) VECTOR-2: This program performs some operations on vectors. It contains OpenMP `parallel` and `for` constructs with a `nowait` clause. The OpenMP code is shown in Fig. 6.

```
int vector2(int A[], int B[], int Y[], int Z[], int n, int m)
{
        int i,j;
        #pragma omp parallel private(A,Z) shared(B, Y)
        {
                #pragma omp for nowait
                for (i=1; i<n; i++){
                        for(j=0; j<n ; j+=20)
                                A[j] = A[j] + 10.25
                        B[i] = (A[i] + A[i-1]) / 2;
                }
                #pragma omp for nowait
                for (i=0; i<m; i++){
                        for(j=0; j<m ; j+=20)
                                Z[j] = Z[j] * 0.025 ;
                        Y[i] = Z[i] * i;
                }
        }
        return 0;
}
```

Fig. 6. OpenMP function to compute vectors using `for` construct.

4.2 Experimental Environment

The experiments have been performed on a 16-node cluster with different computer's configurations. There are two computers with Intel(R) Pentium(R) Dual CPU E2160 at 1.80 GHz, 2 GB of RAM, 5 GB of free HDD; seven computers with Intel(R) Core(TM)2 Duo CPU E7300 at 2.66 GHz, 3 GB of RAM, 6 GB of free HDD; five computers with Intel(R) Core(TM) i3-2120 CPU at 3.30 GHz, 8 GB of RAM, 6 GB of free HDD; and two computers including an AMD Phenom(TM) II X4 925 Processor at 2.80 GHz, 2 GB of RAM, 6 GB of free HDD. All machines are operated by the Ubuntu 14.03 LTS operating system with OpenSSH-Server and MPICH-2. They are interconnected by a 100 Mbps LAN network.

4.3 Experimental Results

Figures 7 and 8 present the execution time in milliseconds for the MAMULT2D program for various size of matrices and different sizes of cluster respectively.

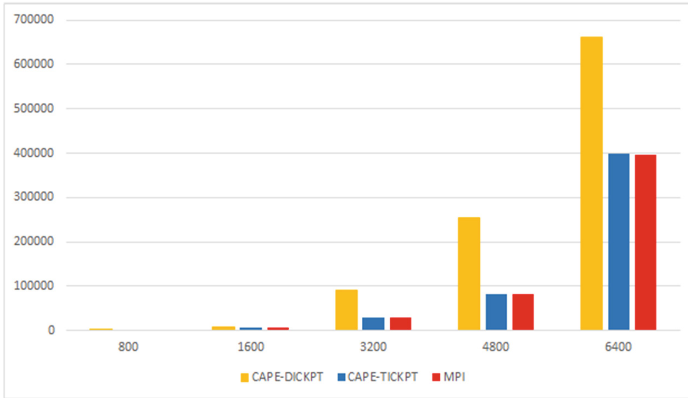

Fig. 7. Execution time (in milliseconds) of MAMULT2D with different size of matrix on a 16-node cluster.

Note that, there are many kinds of processors in different nodes. Some of them include many cores, but a single core was used for each node during the experiments. Three measures are presented at each time: the left one (yellow) for CAPE-DICKPT (the previous version), the middle one (blue) for CAPE-TICKPT (the current version), and the right one (red) for MPI.

Figure 7 presents the execution time for various matrix sizes on a 16-node cluster. The size increases from `800x800` to `6400x6400`. The figure shows that the execution times of all methods are proportional to the matrix size. It also shows that the execution time of CAPE-TICKPT is much lower than the one of CAPE-DICKPT and MPI (around 35%) while the execution time of CAPE-TICKPT and MPI are roughly equal.

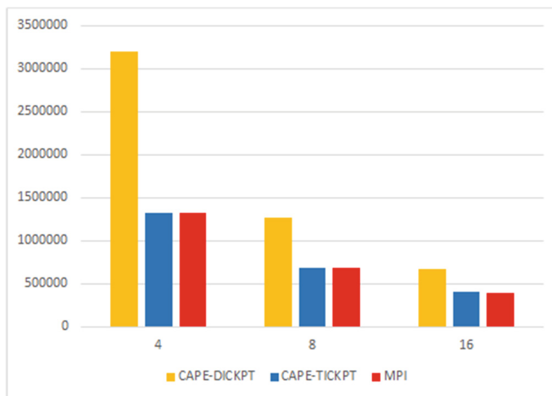

Fig. 8. Execution time (in milliseconds) of MAMULT2D for different cluster sizes.

Figure 8 presents the execution time for a matrix size of `6400x6400` on different cluster size. The number of nodes is successively 4, 8, and 16. The result presented in this figure also shows the similar trend for different matrix size. The execution time of CAPE is significantly reduced so that it is now much closer to an optimized human-written program using MPI.

To demonstrate that the new version of CAPE can run OpenMP programs that do not match with the Bernstein's conditions while achieving high performance, other experiments were conducted and performance were compared with MPI. All of the four other programs presented in Sect. 4.1 have been used to measure the execution time.

Figure 9 presents the execution time in milliseconds of PRIME with $N = 10^6$ on different cluster sizes for CAPE-TICKPT and MPI. It shows that the execution time of MPI is only around 1% smaller than the one of CAPE-TICKPT. In this experiment, the OpenMP `parallel for` directive with the `shared`, `private` and `reduction` clauses are translated and tested for both methods. Table 1 describes the steps executed by the program for both methods. The main different step is the join phase. It gathers the results from all nodes and computes their sum. For the MPI program, the user needs to clearly specify the values that need to be gathered, and then call the `MPI_Reduce()` function after to compute the sum. CAPE-TICKPT automatically identifies the modified value of the shared variables, extracts them into a TICKPT, and then gathers all checkpoints from all the nodes with the `merging` checkpoint operator. However, as the execution time of CAPE-TICKPT is nearly equal to the one of MPI, we consider that we successfully obtained high performance with CAPE.

Table 1. Comparison of the executed steps for the PRIME code for both CAPE-TICKPT and MPI.

Step	CAPE-TICKPT	MPI
Fork	Updates the properties of variables, saves data of shared variables, and re-computes the iterators	Re-computes the iterators
Computation	Computes the divided jobs	Computes the divided jobs
Join	Generates checkpoints, and calls the `merging` checkpoint operator with the `sum` operator	Calls `MPI_Reduce` to gather and sum the results

Figure 10 presents the execution time in milliseconds of PI with a number of steps equal to 10^8 for different cluster sizes using CAPE-TICKPT and MPI. In this experiment, the OpenMP `for` directive with `reduction` clause placed inside the `omp parallel` construct with some clauses are tested. As well as the previous experiments, this figure also shows that CAPE-TICKPT achieves similar performance as MPI.

Figure 11 shows the execution time in milliseconds for the VECTOR-1 program with $N = 10^6$ for different cluster sizes using CAPE-TICKPT and MPI.

In this experiment, OpenMP functions and the `sections` construct with two `section` directives are tested. The figure shows that the larger the number of nodes, the longer the execution time for both methods. The execution time with MPI is smaller than the one of CAPE-TICKPT, but the difference is not significant. Note that there are only two `section` directives in this program, so that both CAPE-TICKPT and MPI distribute the execution to two nodes only. Each node receives and executes the code of a `section`. However, the result has to be synchronized to all nodes on the system. Therefore, the execution time increases when increasing the number of nodes.

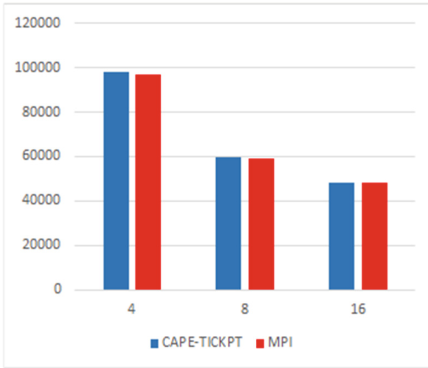

Fig. 9. Execution time (in milliseconds) of PRIME on different cluster sizes.

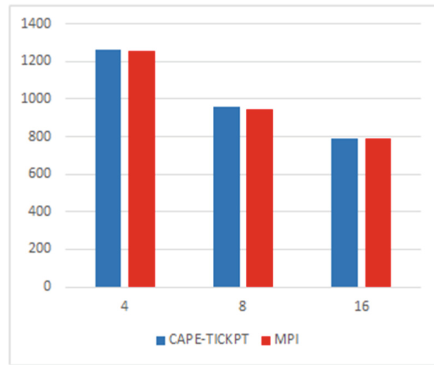

Fig. 10. Execution time (in milliseconds) of PI on different cluster sizes.

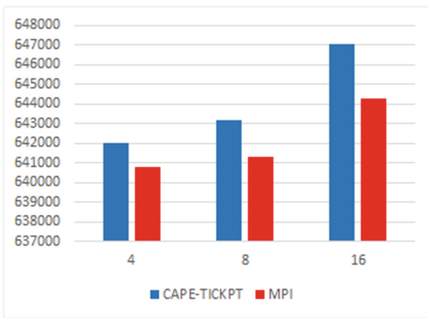

Fig. 11. Execution time (in milliseconds) of VECTOR-1 on different cluster sizes.

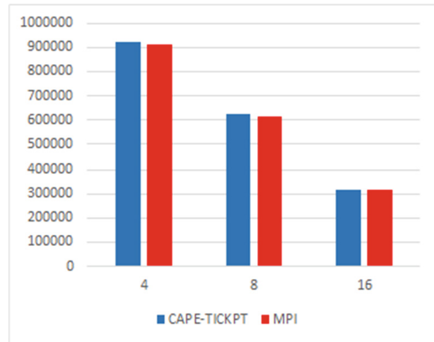

Fig. 12. Execution time (in milliseconds) of VECTOR-2 on different cluster sizes.

Figure 12 shows the execution time in milliseconds for VECTOR-2 with $N = 10^6$ and $M = 1.6 \times 10^6$ on different cluster sizes for both CAPE-TICKPT and MPI. This experiment aims at testing two `omp for` directives with `nowait` clause. The size of the two vectors are different from each other to ensure the nodes take

different time to execute the divided jobs. The execution on each node is marked `nowait` until reaching the end block of the `parallel` region. The figure shows the same trend as the previous experiments. The execution time for CAPE-TICKPT is very close to MPI, the difference being negligible.

5 Conclusion and Future Works

This paper presented the design and implementation of a new execution model and prototypes for CAPE based on TICKPT. With this new capability included, CAPE improves the reliability and can run OpenMP programs that do not require to match the Bernstein's conditions. In addition, the analysis and evaluation of performance of this paper demonstrated that CAPE-TICKPT achieves performance very close to a comparable human-optimized hand-written MPI program. This is mainly due to the fact that CAPE-TICKPT takes benefits of the advantages of TICKPT such as checkpoint operators and can use resources more efficiently. The synchronization phase of the new execution model also avoids the risk of bottlenecks that may have occurred in the previous version.

In the near future, base on this mechanism, we will keep on developing the CAPE framework in order to support other OpenMP constructs. Furthermore, we expect to develop CAPE for GPUs.

References

1. Basumallik, A., Eigenmann, R.: Towards automatic translation of OpenMP to MPI. In: Proceedings of the 19th Annual International Conference on Supercomputing, pp. 189–198. ACM (2005)
2. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. ACM SIGARCH Comput. Archit. News **29**(5), 41–48 (2001)
3. Chen, Z., Sun, J., Chen, H.: Optimizing checkpoint restart with data deduplication. Sci. Program. **2016**, 11 (2016)
4. Cores, I., Rodríguez, M., González, P., Martín, M.J.: Reducing the overhead of an MPI application-level migration approach. Parallel Comput. **54**, 72–82 (2016)
5. Dorta, A.J., Badía, J.M., Quintana, E.S., de Sande, F.: Implementing OpenMP for clusters on top of MPI. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 148–155. Springer, Heidelberg (2005). https://doi.org/10.1007/11557265_22
6. EPCC: EPCC OpenMP micro-benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite
7. Ha, V.H., Renault, E.: Design and performance analysis of CAPE based on discontinuous incremental checkpoints. In: 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (2011)
8. Ha, V.H., Renault, É.: Discontinuous incremental: a new approach towards extremely lightweight checkpoints. In: 2011 International Symposium on Computer Networks and Distributed Systems (CNDS), pp. 227–232. IEEE (2011)

9. Ha, V.H., Renault, E.: Improving performance of CAPE using discontinuous incremental checkpointing. In: 2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC), pp. 802–807. IEEE (2011)
10. Heo, J., Yi, S., Cho, Y., Hong, J., Shin, S.Y.: Space-efficient page-level incremental checkpointing. In: Proceedings of the 2005 ACM symposium on Applied computing, pp. 1558–1562. ACM (2005)
11. Hoeflinger, J.P.: Extending OpenMP to clusters. White Paper, Intel Corporation (2006)
12. Huang, L., Chapman, B., Liu, Z.: Towards a more efficient implementation of OpenMP for clusters via translation to global arrays. Parallel Comput. **31**(10), 1114–1139 (2005)
13. Karlsson, S., Lee, S.-W., Brorsson, M.: A fully compliant OpenMP implementation on software distributed shared memory. In: Sahni, S., Prasanna, V.K., Shukla, U. (eds.) HiPC 2002. LNCS, vol. 2552, pp. 195–206. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36265-7_19
14. Li, C.C., Fuchs, W.K.: Catch-compiler-assisted techniques for checkpointing. In: 20th International Symposium Fault-Tolerant Computing. FTCS-20. Digest of Papers, pp. 74–81. IEEE (1990)
15. Morin, C., Lottiaux, R., Vallée, G., Gallard, P., Utard, G., Badrinath, R., Rilling, L.: Kerrighed: a single system image cluster operating system for high performance computing. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 1291–1294. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45209-6_175
16. OpenMP ARB: OpenMP application program interface version 4.0 (2013)
17. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent checkpointing under unix. Computer Science Department (1994)
18. Renault, É.: Distributed implementation of OpenMP based on checkpointing aided parallel execution. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 195–206. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69303-1_22
19. Sato, M., Harada, H., Hasegawa, A., Ishikawa, Y.: Cluster-enabled OpenMP: an OpenMP compiler for the SCASH software distributed shared memory system. Sci. Program. **9**(2–3), 123–130 (2001)
20. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. Int. J. High Perform. Comput. Appl. **19**(1), 49–66 (2005)
21. Tran, V.L., Renault, É., Ha, V.H.: Improving the reliability and the performance of CAPE by using MPI for data exchange on network. In: Boumerdassi, S., Bouzefrane, S., Renault, É. (eds.) MSPN 2015. LNCS, vol. 9395, pp. 90–100. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25744-0_8
22. Tran, V.L., Renault, E., Ha, V.H.: Analysis and evaluation of the performance of CAPE. In: IEEE International Symposium on 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, pp. 620–627. IEEE (2016)
23. Tran, V.L., Renault, É., Ha, V.H., Do, X.H.: Implementation of OpenMP data-sharing on cape. In: 9th International Symposium on Information and Communication Technology SoICT 2018, pp. 359–366. ACM (2018)
24. Tran, V.L., Renault, É., Ha, V.H., Do, X.H.: Time-stamp incremental checkpointing and its application for an optimization of execution model to improve performance of cape. Informatica **42**(3) (2018)