


New execution model for CAPE using multiple threads on multicore clusters

Xuan Huyen Do¹  | Viet Hai Ha² | Van Long Tran³ | Eric Renault⁴

¹Information Technology Department, University of Sciences, Hue University, Hue, Vietnam

²Office for STIC, University of Education, Hue University, Hue, Vietnam

³Technology and Business Department, Phu Xuan University, Hue, Vietnam

⁴LIGM, University Gustave Eiffel, CNRS, ESIEE Paris, Marne la Vallee, France

Correspondence

Viet Hai Ha, Office for STIC, University of Education, Hue University, Hue, Vietnam.
Email: hviethai@hueuni.edu.vn

Funding information

This research was supported by the Ministry of Education and Training (Vietnam) for project B2019-DHH-09.

Abstract

Based on its simplicity and user-friendly characteristics, OpenMP has become the standard model for programming on shared-memory architectures. Checkpointing-aided parallel execution (CAPE) is an approach that utilizes the discontinuous incremental checkpointing technique (DICKPT) to translate and execute OpenMP programs on distributed-memory architectures automatically. Currently, CAPE implements the OpenMP execution model by utilizing the DICKPT to distribute parallel jobs and their data to slave machines, and then collects the results after executing these distributed jobs. Although this model has been proven to be effective in terms of performance and compatibility with OpenMP on distributed-memory systems, it cannot fully exploit the capabilities of multicore processors. This paper presents a novel execution model for CAPE that utilizes two levels of parallelism. In the proposed model, we add another level of parallelism in the form of multithreaded processes on slave machines with the goal of better exploiting their multicore CPUs. Initial experimental results presented near the end of this paper demonstrate that this model provides significantly enhanced CAPE performance.

KEYWORDS

Checkpointing-aided parallel execution, high-performance computing, OpenMP, parallel computing

1 | INTRODUCTION

OpenMP [1] is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. It consists of a set of compiler directives, library routines, and environment variables that influence runtime behavior. Parallel regions are explicitly specified in an original sequential program by adding compiler directives. By using this programming model, programmers can easily transform a sequential program into a parallel program. Additionally, OpenMP is relatively simple

when compared to other tools such as the message passing interface (MPI) [2].

Based on its easy-to-use and high-performance characteristics, OpenMP has been widely used and has quickly become a standard parallel programming tool for shared-memory systems. Many compilers have been developed for Linux, Windows, MacOS, Solaris, FreeBSD, etc.[3] However, based on its shared-memory model architecture, OpenMP cannot run on distributed-memory systems such as clusters, grids, and clouds.

Many efforts have been made to port OpenMP onto distributed-memory architectures. However, except for

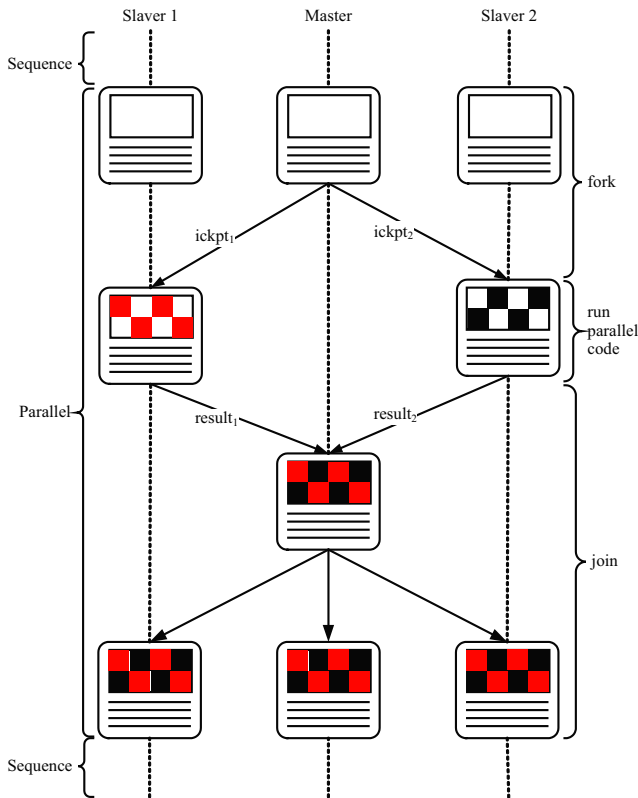


FIGURE 1 Single-threaded CAPE execution model

checkpointing-aided parallel execution (CAPE) [4–10], no solutions have successfully met both of the following requirements: (i) to be fully compliant with the OpenMP API and (ii) to provide high performance. The most prominent approaches include the use of a synchronous serial interface [11], the SCASH library [12], RC model [13], source-to-source translation to a tool such as MPI [14,15], global arrays [16], or clustered OpenMP [17].

CAPE is another effort to overcome the restrictions discussed above based on the checkpointing technique. Two versions of CAPE have been developed and tested on clusters in which one machine (master machine) takes the role of the master thread and the others (slave machines) work as slave threads [7,9]. Its ability to be compliant with the OpenMP standard while providing high performance was proven in [7,8]. However, in the current execution model of CAPE, each slave machine utilizes one process containing a single thread to perform its assigned tasks. Consequently, the advantages of multicore CPU systems are wasted [18]. To overcome this limitation, the most promising solution is to use multithreaded processes on slave machines to execute their tasks in parallel.

In this paper, we propose a novel model to overcome this limitation and present some initial experimental results. The main contributions of this study can be summarized as follow.

- We proposing a novel CAPE execution model that utilizes multiple threads on each slave machine.
- We demonstrate how to implement the proposed model.
- Experimental results demonstrate a performance improvement rate of 1.6 to 3.1 times compared to previous methods depending on the number of CPU cores used.

The remainder of this paper is organized as follows. Section 2 discusses the principles of CAPE, ranging from its execution model and method of compiling OpenMP programs to its limitations. The next section presents the proposed model for CAPE on multicore CPU systems. Section 4 presents experimental results and evaluations in which the proposed execution model achieves a speedup ranging from 1.6 to 3.1 times compared to the previous model of CAPE. The final section discusses our conclusions and plans for future work.

2 | PRINCIPLES OF CAPE

2.1 | Execution model

The first important difference between CAPE and the original OpenMP model is the use of processes instead of threads. This replacement facilitates the distribution of parallel sections of OpenMP programs onto networked machines.

Figure 1 illustrates the execution model of CAPE while running on a system consisting of three machines: one master and two slaves. The master is in charge of the initial threads from the original OpenMP execution model, and the two slaves execute threads assigned by the master. First, an application is initialized and runs on all nodes until reaching a parallel section. At this point, the master divides the work of the parallel section and distributes it to the slave nodes utilizing discontinuous incremental checkpointing (DICKPT) [4,7]. Each slave node receives a separate checkpoint, injects it into its memory space, and executes the divided work from the parallel section. Next, the calculated results are extracted according to the checkpoints and sent to the master node. The master node receives all checkpoints from slave nodes and merges them into a single united checkpoint, which is then broadcasted to the system. After every node (including the master node) injects the united checkpoint into its memory space, they can be considered to be in the same state of completing the parallel section. They then continue the next instruction in the application.

It should be noted that in this execution model, CAPE can only be applied to OpenMP programs matching Bernstein's conditions [19]. However, the introduction of some additional processes for OpenMP shared variables that do not change the CAPE principle can overcome this restriction.

2.2 | Compiling OpenMP programs for execution on the CAPE platform

The compiling chain for CAPE is illustrated in Figure 2. CAPE contains a compliant tool to translate and execute OpenMP programs on a networked machine, such as a cluster, automatically. The CAPE compiler utilizes a set of transformation prototypes to transform an original OpenMP program (such as the code in the upper rectangle in Figure 3) into a CAPE program (such as the code in the lower rectangle in Figure 3), which only contains C/C++ instructions. Therefore, the CAPE program can be compiled into an executable form by a typical C/C++ compiler and can then be executed on networked machines supporting the CAPE platform.

2.3 | Prototypes to transform OpenMP programs into CAPE programs

In the CAPE framework, a set of functions has been defined and implemented to perform the tasks related to DICKPT, including distributing checkpoints, sending/receiving checkpoints, and extracting/injecting checkpoints from/to program memory. Additionally, a set of transformation prototypes (templates) are defined in the CAPE compiler to perform the translation of OpenMP programs into CAPE programs automatically and make such programs executable in the CAPE framework. Thus far, nested loops and shared-data variable constructs have not been supported. However, this is not a significant issue because it can be resolved at the level of source-to-source translation and does not require any modifications to the CAPE philosophy. After being translated, the original OpenMP source code is free of OpenMP directives and structures. Figure 3 presents an example of code substitution for the specific case of the **omp parallel for** construct. This is a representative example of the substitutions we implemented for other OpenMP constructs [20].

The automatically generated code is based on the following functions, which are components of the CAPE framework:

startt () begins monitoring the checkpointing application process;

stop() begins monitoring the application process;

create (file) generates a discontinuous incremental checkpoint and saves it to a file;

inject (file) injects a checkpoint stored in a file into the memory space of the monitored process;

send (file, node) sends the checkpoint stored in a file from the current node to another node;

waitfor (file) waits for the checkpoint file;

merge (file1, file2) merges the checkpoint in file2 into that in file1;

broadcast (file) sends a checkpoint file to all slave nodes;

receive (file) waits for a checkpoint and stores it into a file.

lastparallel() returns TRUE when the current parallel block is the last block in the entire program or returns FALSE otherwise.

2.4 | Drawbacks of CAPE on multicore systems

Currently, multicore architectures are very common in CPUs, and the number of cores is increasing steadily. Commodity computers are equipped with two to eight CPU cores. This has led to the popularity of computer networks with multicore nodes. Therefore, high-performance computing tools should maximally exploit this architecture. However, in the current execution model, CAPE does not focus on this architecture when transforming a parallel section in an application into many sequential parts and assigning these parts to slave nodes. Therefore, there is only one sequential process for a user application running on each slave node, which wastes the computing resources of multicore machines. This principle is

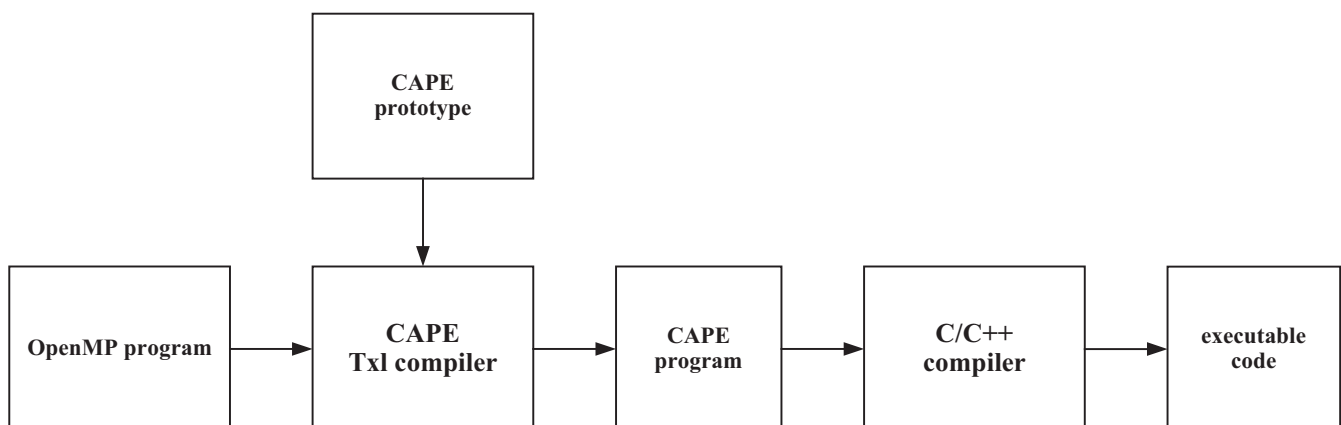


FIGURE 2 Steps to compile OpenMP programs in CAPE

clearly illustrated in Figure 4, where the slave machine has a quad-core CPU, but only one core is fully exploited, while the other cores are virtually idle.

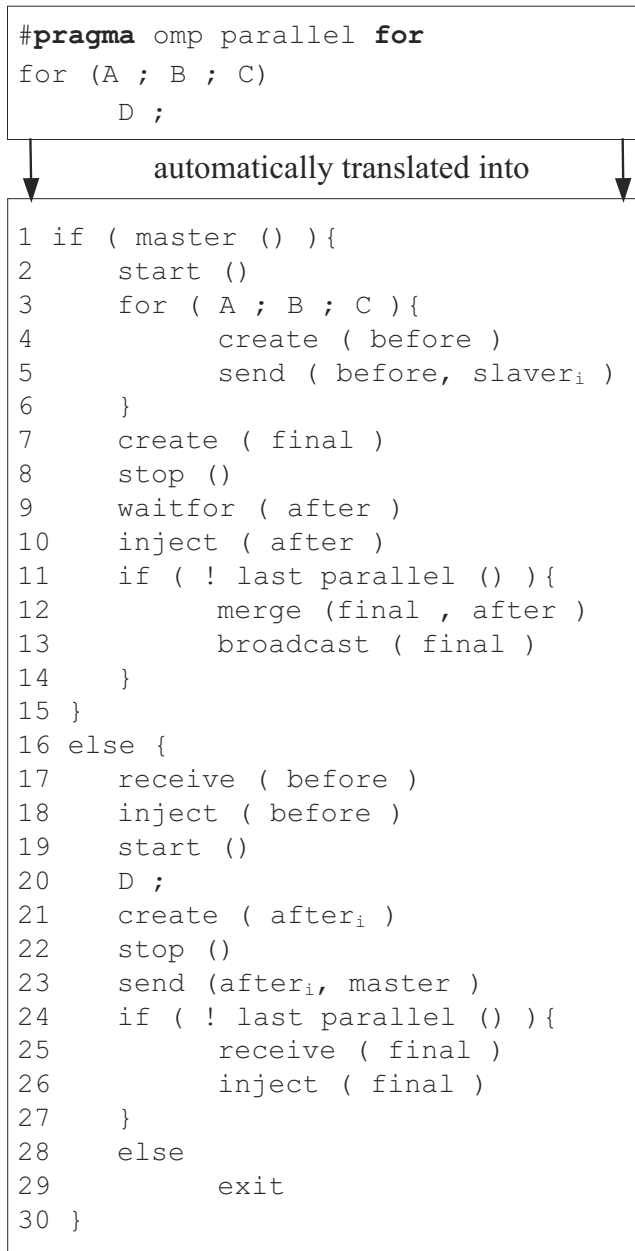


FIGURE 3 Prototype for translating pragma omp parallel for sections

3 | NOVEL CAPE EXECUTION MODEL FOR EXPLOITING THE ABILITIES OF MULTICORE SYSTEMS

Our straightforward concept for better exploiting the calculation resources of multicore systems is to increase the number of processes/threads that run in parallel on each node in a system, particularly slave nodes. The details of our solutions for implementing this concept are presented below.

3.1 | Novel CAPE execution model using two parallel levels

In this solution, each original OpenMP parallel section is divided into many multithreaded processes, each of which is executed on a node, similar to the current CAPE execution model. Therefore, on each node, applications run on slave machines in multithreaded processes, which can better exploit the performance of multicore CPU systems. The main problem with this solution is the ability to manage multithreaded applications using the checkpointer in CAPE. Many conflicts occur when a single-threaded checkpointer defines the memory space of a multithreaded process in a write-protected state to perform checkpointing tasks. Therefore, it is necessary to upgrade the current checkpointer of CAPE, which can only checkpoint the single threads, to a multithreaded checkpointer.

Figure 5 illustrates the proposed multithreaded CAPE execution model. In this model, each original OpenMP parallel section is divided into many processes, each of which is executed on a node, similar to the current CAPE execution model. However, in the proposed model, the processes on slave nodes are multithreaded, unlike in the current execution model. Therefore, the application processes on slave nodes can better exploit the performance of multicore CPU systems. Therefore, we have a two-level parallel execution model in which level 1 is related to the division and simultaneous execution of original OpenMP parallel tasks on multiple machines and level 2 is related to the use of multiple threads on each slave node to execute its tasks in parallel.

CPU History

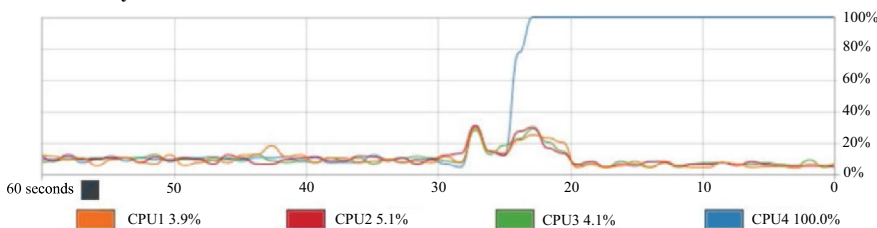


FIGURE 4 Ratio of exploiting cores when executing one process on each calculation node

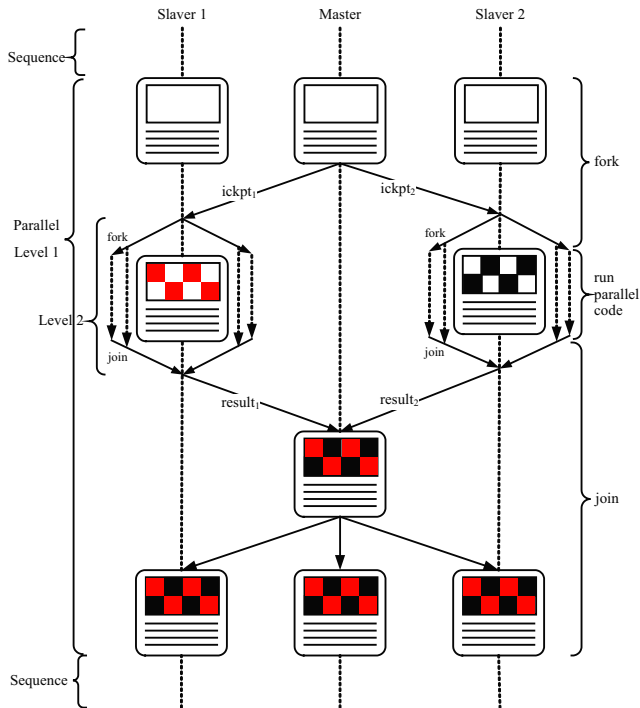


FIGURE 5 Multithreaded CAPE execution model

3.2 | Prototype for translating omp parallel for for construction into multithreaded CAPE

To compile and execute OpenMP constructors in the proposed CAPE execution model, the transformation prototypes of the compiler must also be upgraded. For example, as shown in Figure 6, the original OpenMP **omp parallel for** construction is compiled into a set of instructions containing both OpenMP and CAPE functions. This provides the ability to execute this construction in two parallel levels, where the first distributes tasks onto multiple machines and the second handles the use of multiple threads to execute divided tasks in parallel.

In Figure 6, lines 2 to 13 are executed on the master node, whereas lines 17 to 31 are executed on slave nodes. On the master node, at line 2, the memory of the application is set to be write protected and checkpointing is initialized. Lines 4 to 6 are used to send checkpoints to the slave machines to initialize the application state and assign tasks from the for loop. Next, the master node creates a checkpoint to extract local results and stops the checkpointing process. Lines 9 and 10 are used to receive the execution results from all slave nodes and inject them into the memory of the application. If there are other OpenMP parallel instructions, the master node merges the results from the slave nodes with the local results and broadcasts the final results to all slave nodes to synchronize their memory spaces. The master node then prepares for the next instructions.

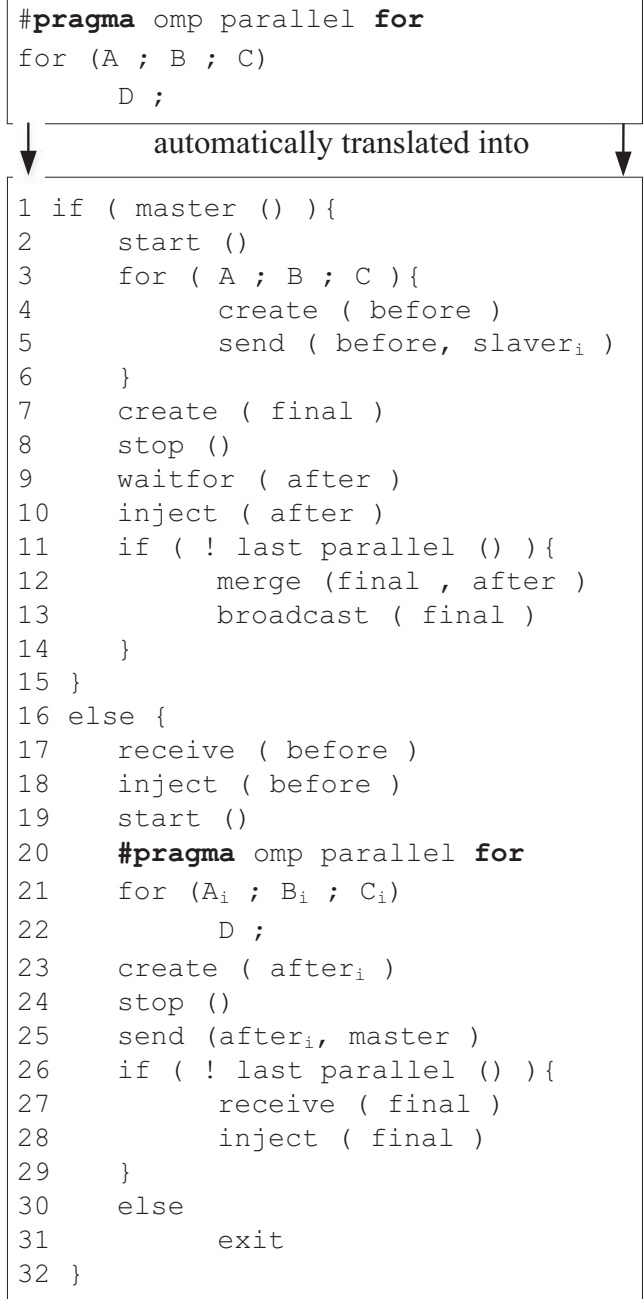


FIGURE 6 Prototype to translate pragma omp parallel for sections into multithreaded CAPE

Regarding the slave nodes, in lines 17 to 19, each node receives a checkpoint and injects its values into the application memory. This phase serves to initialize the memory space and prepare each node to receive information regarding how to execute its assigned processes from the **omp parallel for** construct. Subsequently, the checkpointing status is updated before executing the **omp parallel for** construct in lines 20 to 22. Because this is a real OpenMP parallel construction, it will be executed by multiple threads in parallel. This construct is similar to the **omp parallel for** construct in the original program, except for the number of iterations, which is equal

to the quotient of the original number of iterations and the number of slave nodes. Therefore, in this section, OpenMP is responsible for dividing tasks between threads and for execution and synchronization between threads. Because it is a standard tool for multithreaded programming, OpenMP can handle these jobs effectively. Therefore, CAPE only has to perform the work of monitoring and checkpointing. This process is defined in lines 23 to 25, where CAPE extracts local execution results from the current node and sends them to the master node. The slave node then receives the merged results of all other slave nodes from the master node and injects them into the application memory space, and then switches to execute the next instruction in the application.

3.3 | Challenges of multithreaded checkpointing

Switching to a two-level parallel execution model requires a new checkpointer that can checkpoint multithreaded processes. This is necessary to satisfy the principles of CAPE, which is based on the checkpointing technique. The tasks of dividing and distributing the tasks in a parallel section from the master node to the slave nodes and then extracting the results of executing divided sections on the slave nodes for the master node are performed using snapshots (checkpoints) of the application's memory. One important compulsory condition for this mechanism is that the data region of the application must be the same for all nodes. This means that if the value of variable A in the application running on the master node is stored at address M1, then the address of this value on all slave nodes must also be M1. Most operating systems and CPUs that support virtual memory management mechanisms divide the memory space of an application into equal-sized pages starting at an index of zero [21,22]. Therefore, if one program runs on the same operating system on different computers, then in the memory spaces of that program on each computer, the addresses of the data regions will be the same (some operating systems set different shift spaces at the beginnings of data regions, but this can be circumvented by setting a system option before starting the application). For example, on Linux 32 bit operating systems, each running program is allocated a virtual (logical) memory space of 4 GB, where 1 GB is used for kernel space and 3 GB are used for user space. This virtual memory is organized into a series of continuous memory pages of equal size (4 KB). However, the condition that the program data memory spaces have the same addresses is no longer fully satisfied when running multiple threads. When a new thread is created, its local data are allocated to the stack region of the main process. On Linux/x86-32 systems, the default stack size for a new thread is 2 MB [23]. Consequently, it is necessary

to synchronize the addresses of data in the stack regions of the threads when processing checkpoints.

The current checkpointer in CAPE uses the **ptrace** mechanism [24,25] to monitor and execute checkpointing tasks. This is the typical mechanism for the checkpointing technique. However, it cannot be applied to the monitoring of multithreaded processes. To solve this problem, in our new checkpointer, we utilize a checkpointing library and insert its checkpointing functions inside the target application. This does not alter the main principles of CAPE. We also change the method of locking (setting to write-protected status) the memory spaces of applications from using a kernel-level driver to directly locking each memory region using the **mprotect** function in the user-level space, which reduces potential errors and decreases total execution time.

3.4 | Theoretical speedup of the proposed CAPE execution model

According to Amdahl's law [26], the theoretical speedup when using multiple processors can be expressed as

$$S_{\text{latency}} = \frac{1}{(1-p) + \frac{p}{s}}, \quad (1)$$

Where S_{latency} is the theoretical speedup of the execution of the entire task, s is the speedup of the parts of the task that benefit from improved system resources (number of parallel processes or threads), and p is the proportion of the execution time that the parts benefiting from improved resources originally required.

Consider the case of executing a parallel application on a system of eight machines, each of which is equipped with a dual-core CPU. Under the most ideal assumptions, the code that is executed in parallel takes up the entire duration of the program ($P = 1$) and the data communication time between machines is ignored. Then, the maximum acceleration factor of CAPE in the case of using only one core (single-threaded processes) is

$$S_{\text{latency}} = \frac{1}{(1-1) + \frac{1}{8}} = 8. \quad (2)$$

In the case of using both cores of each CPU (two-threaded processes), the maximum acceleration factor is

$$S_{\text{latency}} = \frac{1}{(1-1) + \frac{1}{8*2}} = 16. \quad (3)$$

Therefore, the greatest theoretical speedup factor when using two cores compared to that when using one core is a factor of. A speedup factor of 16 is achieved compared to the case of sequential application. Actual execution times and speedup factors were derived experimentally and the results are

presented in Section 4. It should be noted that for parallel programs, in addition to the time spent on dividing, transmitting, and synthesizing results, there are some other factors affecting system performance, such as time required to create and manage processes/threads. All of these factors always make the real speedup of parallel programs less than the theoretical value.

4 | EXPERIMENTS AND EVALUATIONS

To measure the impact of the proposed execution model on system performance, following the mathematical analysis presented in Section 3, various experiments were conducted. These experiments were performed using a matrix-matrix multiplication program on two clusters of personal computers connected through a 100 Mbps local area network and operated by the Ubuntu 18.04.3 LTS operating system with the following OpenSSH server configurations.

1. A 16 node cluster with different computer configurations. There were six Intel(R) Core(TM) i3-2100 3.1 GHz, 8 GB RAM computers, two AMD Phenom(TM) II X4 925 2.80 GHz, 8 GB RAM computers, one Intel(R) Core(TM) i3-2120 3.3 GHz, 4 GB RAM computer, two Intel(R) Pentium(R) Dual E2160 1.80 GHz, 2 GB RAM computers, and five Intel(R) Core(TM)2 Duo CPU E7300 2.66 GHz, 3 GB RAM computers.
2. An eight-node cluster with different computer configurations. There were six Intel(R) Core(TM) i3-2100 3.1 GHz 8 GB RAM computers and two AMD Phenom(TM) II X4 925 2.80 GHz, 8 GB RAM computers. Tests were executed with different matrix sizes of 9600*9600 and 6400*6400, and different numbers of threads on the slave nodes of one, two, and four. Each scenario was tested at least 10 times to measure total execution time, and a confidence interval of at least 98% was achieved for our measurements. The data reported here represent the average values of the 10 measurements. Two comparisons were used to evaluate the performance of the proposed execution model. The first was a comparison to the original execution model, which uses only a single thread on slave nodes. The second was a comparison to the MPI, which is the best-performing tool for parallel execution on distributed machines. Figures 7 and 8 demonstrate that both the CAPE and MPI run times are reduced when the numbers of threads/processes increase. In all cases, MPI performance is better than CAPE performance with a relatively stable ratio ranging from 2% to 21%. This is reasonable because in CAPE, applications are monitored and checkpointed, which increases their run times. It should be noted that relative to MPI, the performance of CAPE is very promising.

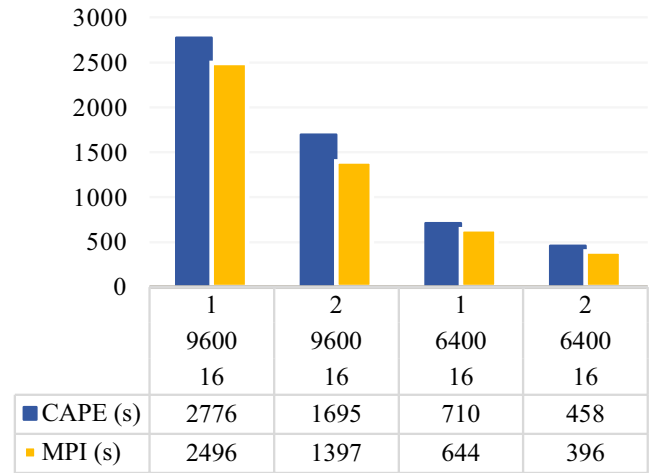


FIGURE 7 Execution times (in seconds) on a 16 node cluster with different numbers of threads

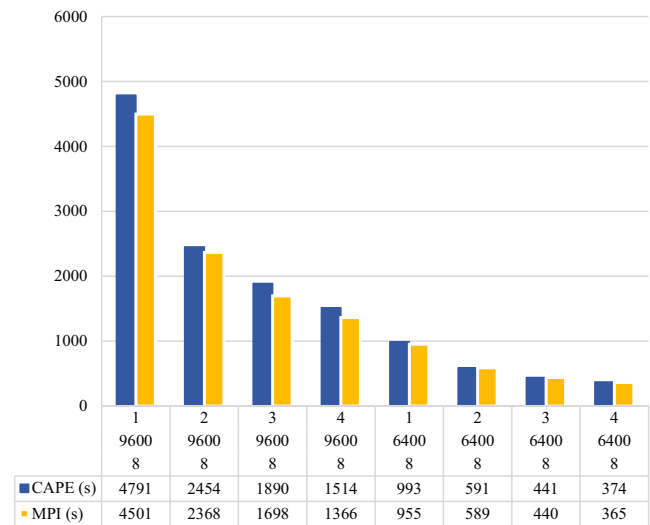


FIGURE 8 Execution times (in seconds) on an eight-node cluster with different numbers of threads

Table 1 reveals that the performance of the original CAPE version using checkpointing drops by approximately 15% on average in the case of 16 nodes and 5% in the case eight nodes compared to the MPI. The speedup of CAPE compared to running OpenMP on one node (a machine with a quad-core CPU) ranges from 6.77 to 7.68 times in case of eight nodes and from 10.27 to 13.09 times in the case of 16 nodes. In the case of the 9600*9600 matrix size on the eight-node cluster, CAPE performance is 4% to 11% lower than that of the MPI.

Figure 9 presents the CAPE and MPI speedup ratios for a variety of matrix sizes and numbers of threads when running on an eight-node cluster. One can see that OpenMP's speedup ratio is very stable because it only runs on one computer. CAPE's speedup ratio is roughly equivalent to that of

TABLE 1 Performance comparisons between CAPE and the MPI

Num of nodes	Size	Core num	CAPE (s)	MPI(s)	OpenMP(s)	MPI/CAPE	OpenMP/CAPE
a	b	c	d	e	f	$g = (1-d/e) * 100$	$h = f/d$
16	9600	1	2776	2496	32 556	-11%	11.73
16	9600	2	1695	1397	17 401	-21%	10.27
16	6400	1	710	644	9292	-10%	13.09
16	9600	2	458	396	4919	-16%	10.74
Average						-15%	11.46
8	9600	1	4791	4501	34 991	-6%	7.30
8	9600	2	2454	2368	17 942	-4%	7.31
8	9600	3	1890	1698	13 011	-11%	6.88
8	9600	4	1514	1366	10 243	-11%	6.77
8	6400	1	993	955	7147	-4%	7.20
8	6400	2	591	589	4488	0%	7.59
8	6400	3	441	440	3386	0%	7.68
8	6400	4	374	365	2653	-2%	7.09
Average						-5%	7.23

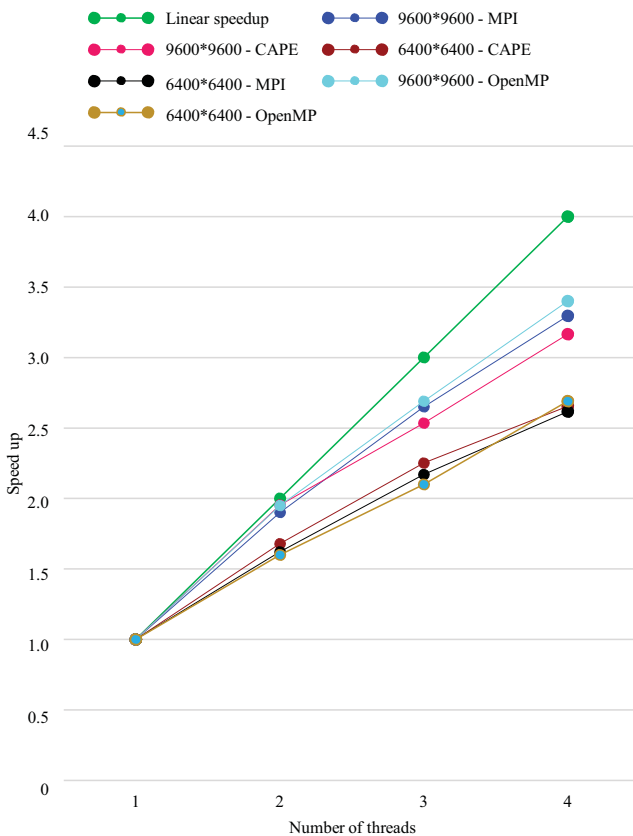


FIGURE 9 Speedup vs number of threads on an eight-node cluster

the MPI. For a smaller matrix size, the speedup is lower because the time required for task division and data synchronization occupies a greater proportion of the total run time. When the matrix size is larger, the speedup is higher because

the computation time occupies a greater proportion than the time required for task division and data synchronization. The speedup ratios of CAPE and the MPI are the closest with the linear speedup in the case of utilizing two threads/processes (1.90 to 1.95 times). In the cases of using three or four threads/processes, the speedup ratios increase, but not exactly linearly relative to the number of threads/processes. It should be noted that the Intel(R) Core(TM) i3-2100 CPU only has two real cores (two hyperthreaded CPUs). This is why we only tested with a number of threads limited to four, which is equal to the number of cores multiplied by the hyperthreading factor of each core. Results with more threads have been omitted because we did not observe any performance increases in cases with larger numbers of threads. This is reasonable because in such a setting, a program is essentially executed in the form of OpenMP code, meaning the optimal number of threads is equal to the number of cores multiplied by the hyperthreading factor of each thread, as discussed in [27]. Therefore, we can conclude that the speedup ratio of CAPE increases in the proposed execution model and that the speedup is approximately linear relative to the number of threads as long as this number is less than or equal to the number of cores on the CPU. This is the most important result demonstrating the advantages of the proposed execution model.

5 | CONCLUSIONS AND FUTURE WORK

This paper presented the design and experimental results of a novel execution model for CAPE that utilizes two-level

parallelism to implement OpenMP on distributed machines. Running multiple threads on slave nodes that are equipped with multicore CPUs can increase the performance of CAPE, where the greatest speedup ratio is achieved when the number of threads is equal to the number of cores in the CPUs. Additionally, the speedup of the proposed execution model is almost equal to that of the MPI, which is the best-performing tool for parallel execution on distributed machines. All of these factors demonstrate the advantages of the proposed execution model when utilizing multicore systems.

In the near future, we will conduct additional experiments with CAPE on other applications and machines equipped with CPUs containing additional cores.

ACKNOWLEDGMENTS

We would like to express our sincere thanks to the Ministry of Education and Training of Vietnam for funding this research.

ORCID

Xuan Huyen Do  <https://orcid.org/0000-0001-7108-537X>

REFERENCES

- OpenMP.org, Openmp application programming interface, version 4.5, 2015, Mar. 2020, available at: <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>.
- MPI Forum, Mpi: A message-passing interface standard, version 3.1, 2015, Mar. 2020, available at <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.
- OpenMP.org, Openmp compilers and tool, 2019, Mar. 2020, available at <https://www.openmp.org/resources/openmp-compilers-tools/>.
- H. V. Hai and R. Éric, *Discontinuous incremental: A new approach towards extremely lightweight checkpoints*, in Proc. Int. Symp. Comput. Netw. Distrib. Syst. (CNDS 2011) (Tehran, Iran), Feb. 2011, pp. 227–232.
- H. V. Hai and R. Éric, *Improving performance of cape using discontinuous incremental checkpointing*, in Proc. IEEE Int. Conf. High Perform. Comput. Commun. (HPCC 2011) (Alberta, Canada), Sept. 2011, pp. 802–807.
- H. V. Hai and R. Éric, *Design of a shared-memory model for cape*, in Proc. Int. Workshop on OpenMP (IWOMP 2012) (Rome, Italy), June. 2012, pp. 262–266.
- H. V. Hai and R. Éric, *Design and performance analysis of cape based on discontinuous incremental checkpoints*, in Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal Process. (PacRim 2011) (BC, Canada), Aug. 2011, pp. 862–867.
- T. V. Long, R. Éric, and H. V. Hai, *Analysis and evaluation of the performance of cape*, in Proc IEEE Int. Conf. Scalable Comput. Commun. (ScalCom 2016) (Toulouse, France), July 2016, pp. 620–627.
- T. V. Long et al., *Design and implementation of a new execution model for cape*, in Proc. Int. Symp. Inform. Commun. Technol. (SoICT's 2017) (Nha Trang, Vietnam), Dec. 2017, pp. 453–459.
- T. V. Long et al., *Time-stamp incremental checkpointing and its application for an optimization of execution model to improve performance of cape*, Informatica **43** (2018), 301–311.
- D. Margery et al., *Kerrighed: A SSI cluster OS running OpenMP*, in Proc. European Workshop OpenMP (EWOMP 2003) (Aachen, Germany), 2003.
- Y. Ojima et al., *Performance of cluster-enabled openmp for the scash software distributed shared memory system*, in Proc. IEEE/ACM Int. Symp. Clust. Comput. Grid (CCGRID'03) (Tokyo, Japan), May 2003, pp. 450–456.
- S. Karlsson, S.-W. Lee, and M. Brorsson, *A fully compliant OpenMP implementation on software distributed shared memory*, in High Performance Computing—HiPC 2002. Springer, Berlin, Heidelberg, 2002, pp. 195–206.
- A. Saa-Garriga, D. Castells-Rufas, and J. Carrabina, *Omp2mpi: Automatic mpi code generation from openmp programs*, in Proc. Workshop High Perform. Energy Effic. Embed. Syst. (HIP3ES), Netherlands, Amsterdam), Jan. 2015.
- A. C. Jacob et al., *Exploiting fine- and coarse-grained parallelism using a directive based approach*, in Proc. Int. Workshop OpenMP (IWOMP 2015) (Aachen, Germany), Oct. 2015, pp. 30–41.
- L. Huang, B. Chapman, and Z. Liu, *Towards a more efficient implementation of OpenMP for clusters via translation to global arrays*, Parallel Comput. **31** (2005), 1114–1139.
- J. P. Hoeinger, *Extending OpenMP to clusters*, White Paper, Intel Corp., 2006, available at http://www.classcloud.org/grid/raw-attachment/wiki/Osaka/Intel_Extend_OpenMP_Cluster.pdf.
- H. V. Hai et al., *Creating an easy to use and high performance parallel platform on multi-cores networks*, in Proc. Mob., Secur. Programmable Netw. (MSPN 2016) (Paris, France), June 2016.
- A. J. Bernstein, *Analysis of programs for parallel processing*, IEEE Trans. Electr. Comput. **EC-15** (1966), no. 5, 757–763.
- J. M. Dorta et al., *Implementing OpenMP for clusters on top of mpi*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2005, Springer, Berlin, Heidelberg, 2005, pp. 148–155.
- M. Gorman, *Understanding the Linux Virtual Memory Manager*, in Understanding The Linux Virtual Memory Manager Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
- Intel Inc., *5-level paging and 5-level ept*. white paper. revision 1.1, 2017, Oct. 2019, available at https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf.
- Canonical Ltd, *Ubuntu manpage: Pthread_create—create a new thread*, Oct. 2019, available at http://manpages.ubuntu.com/manpages/bionic/man3/pthread_create.3.htm.
- P. Padala, *Playing with ptrace, part I*, 2002, Oct. 2019, available at <https://www.linuxjournal.com/article/6100>.
- P. Padala, *Playing with ptrace, part II*, 2002, Oct. 2019, available from <https://www.linuxjournal.com/article/6210>.
- G. M. Amdahl, *Validity of the single-processor approach to achieving large scale computing capabilities*, 1967, Oct. 2019, available at <http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf>.
- J. H. Abdel-Qader and R. S. Walker, *Performance evaluation of openmp benchmarks on intel's quad core processors*, in Proc. WSEAS Int. Conf. Comput.: Part of the 14th WSEAS CSCC Multiconf., vol. 1, (Stevens Point, WI, USA), July 2010.

AUTHOR BIOGRAPHIES



Do Xuan Huyen received his MS degree in Information Technology from the Asian Institute of Technology, Thailand, in 2003. He is currently a PhD student of Computer Science at the University of Sciences, Hue University, Vietnam. His current research interests include the development of CAPE and E-government.



Ha Viet Hai received his PhD from the Institut Mines Télécom, Télécom SudParis, France, in 2012. He currently works at the University of Education, Hue University, Vietnam, where he is a lecturer in the Informatics department and the head of the Office for Science, Technology, and International Cooperation. His current research interests include the development of CAPE and the application of ICT in teaching.



Tran Van Long received his PhD degree from Télécom SudParis, France, in 2018. He currently works at Phu Xuan University, Vietnam, where he is the dean of the Technology and Business Department. His current research interests include the development of CAPE and artificial intelligence.



Éric Renault received his MS degree in Computer Engineering (diplôme d'ingénieur) from ISTY and a second MS degree in Computer Science (DEA) from UVSQ in 1995. He received a PhD in Computer Science from UVSQ in 2000 and a qualification for advising PhD students (HDR) from UPMC in 2011. He is a full professor at ESIEE Paris and a member of LIGM (UMR CNRS 8049) at the Université Gustave Eiffel, France. His research interests include high-performance computing and messaging, compilation, virtualization, positioning, and lightweight security for mobile, sensor, and vehicular networks. He has worked on several European projects and served as an expert for the evaluation of French national projects (ANR), private company research works (MESRI), and Eiffel Excellence Scholarship Program applications (Campus France). He has authored more than 100 articles in international journals and conferences.