Implementation of OpenMP Data-Sharing on CAPE

Van Long Tran
SAMOVAR, Télécom SudParis, CNRS,
Université Paris-Saclay
Évry, France
Hue Industrial College
Hue, Vietnam
van_long.tran@telecom-sudparis.eu

Éric Renault
SAMOVAR, Télécom SudParis, CNRS,
Université Paris-Saclay
Évry, France
eric.renault@telecom-sudparis.eu

Xuan Huyen Do
College of Sciences, Hue University
Hue, Vietnam
doxuanhuyen@gmail.com

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

ABSTRACT

CAPE (Checkpointing-Aided Parallel Execution) is a framework that automatically translates and executes OpenMP on distributed-memory architectures based on checkpoint technique. In some experiments, this approach shows high-performance on distributed-memory system. However, it has not been fully developed yet. This paper presents an implementation of OpenMP data-sharing on CAPE that improves the capability, reduces checkpoint size and makes CAPE even more performance.

CCS CONCEPTS

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

KEYWORDS

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

ACM Reference Format:
Van Long Tran, Éric Renault, Xuan Huyen Do, and Viet Hai Ha. 2018. Implementation of OpenMP Data-Sharing on CAPE. In *The Ninth International Symposium on Information and Communication Technology (SoICT 2018), December 6–7, 2018, Danang City, Viet Nam.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3287921.3287950

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoICT 2018, December 6–7, 2018, Danang City, Viet Nam
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6539-0/18/12…$15.00
https://doi.org/10.1145/3287921.3287950

1 INTRODUCTION

OpenMP [18] and MPI [17] are two widely-used tools that are provided to develop parallel applications. OpenMP has been developed for shared-memory architectures, while MPI is a tool for high-performance computing on distributed-memory environments. Compared to MPI, OpenMP is easier to use. It is due to OpenMP automatically executes a sequential program in parallel based on adding the OpenMP constructs, clause, and runtime functions while MPI request 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, excluding CAPE [8][10][23][25], 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 [16], SCASH [22], the use of the RC model [14], performing a source-to-source translation to a tool like MPI [2] or Global Array [12], or Cluster OpenMP [11].

In OpenMP, a relaxed-consistency (RC) shared-memory model is implemented. All OpenMP threads can access to the same place in the memory to store and retrieve variables, this memory is called shared memory. Besides, each thread is allowed to have its own memory, called local memory. The local memory provides a temporary view of memory. On the one hand, it allows the threads to cache the shareable variables and thereby to avoid going to shared memory for every access to a variable. The updating of shareable variables from local memory to shared memory is only mandatory requirement at the synchronization points. On the other hand, local memory allows the threads to store the private variables accessed only by the owner. The shared or private properties of variables determines the updating of shared memory. In OpenMP, it not only has a set of rules to identify the variables property implicitly, but also provides `threadprivate` construct and a set of clauses that allow the programmers to set property for variables explicitly.

In the previous works, CAPE have implemented the RC shared-memory model of OpenMP on distributed-memory architectures. The shared and private variables of the program are stored in the memory of different nodes and independently accessed by the owner. When the program reaches the synchronization points, these data are synchronized by using DICKPT [9]. However, this implementation has not handled data-sharing attributes yet, all

modification of memory including private data at each node is extracted to store into a DICKPT. These checkpoints from all slave nodes are sent to and merged at the master node at the join phase. Then, the merged checkpoint is distributed to all slave nodes to update application memory. This implementation does not only transfers unnecessary data, but also makes the program work incorrectly.

To solve these issues, in new design and implementation of CAPE based on Timestamp Incremental Checkpointing (TICKPT), only the modified data of shared variables is selected to synchronize. In order to do that, the variables attribute are handled follow by OpenMP data-sharing attribute rules. In addition, checkpoints are merged using the operations on checkpoints that significantly reduce checkpoint sizes and execution time. This paper present the method to handle data-sharing attribute for variables in CAPE based on TICKPT (CAPE-TICKPT).

The paper is organized as follows: the next Section presents the related works on CAPE and checkpoint techniques. Then Section 3 presents a reminder of OpenMP data-sharing attribute. The mechanism to implement the OpenMP data-sharing for CAPE is presented and evaluated in Section 4 and 5. Finally, Section 6 concludes the papers.

2 RELATED WORKS

2.1 CAPE

At this moment, two versions of CAPE have been developed. The first one using complete checkpoints [21][15] and the second one using incremental checkpoints [8][10]. The second one is named CAPE-2 that demonstrated high performance and complete compatibility with OpenMP [10] [25].

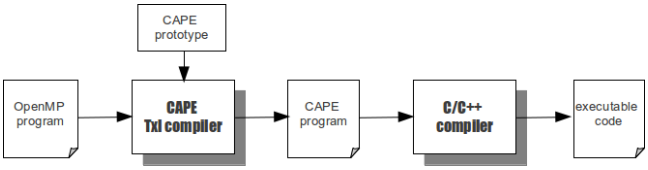

Figure 1: Translation OpenMP programs with CAPE

Basically, in order to execute OpenMP program on distributed-memory system, CAPE uses a set of templates to translate OpenMP source code to CAPE source code, and then, CAPE source code will be compiled by C/C++ compiler to execute code. This code can be executed on distributed-memory systems under the support of CAPE framework. The different steps of CAPE compilation process for C/C++ OpenMP program are shown in the Figure. 1. To explain more about CAPE principles, we will discuss the execution model that CAPE framework is given to execute CAPE source code on distributed-memory systems, and discuss transformation of prototypes in the next part of this section.

2.1.1 Execution model. The ideal of the CAPE execution model is used for checkpoints to implement the OpenMP fork-join model. This mechanism is shown in Fig. 2.

In order to execute a CAPE code on distributed-memory architectures, the program first run on set of nodes on the system, each

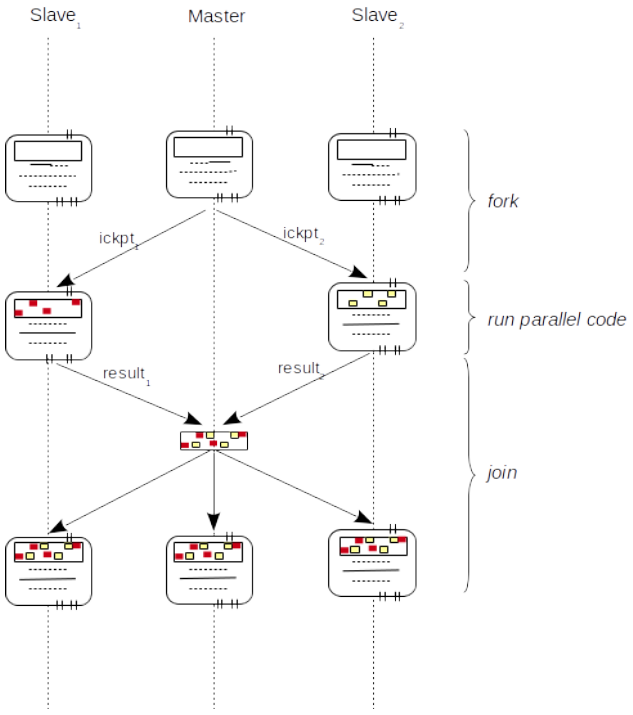

Figure 2: CAPE execution model

node works as a process. Whenever the program meets a parallel section, the master distributes jobs to the slaves process by using Discontinuous Incremental Checkpoints (DICKPT) [8][9]. By this approach, the master node generates DICKPTs and sends them to slave nodes, each slave node will receive a checkpoint. After sending checkpoints, the master will wait for the returned results from slaves. The next step of master is that they receive and merge the result of the checkpoints before injecting this into its memory. For slave nodes, they receive different checkpoints, and then, they inject this checkpoint into their memory to do the divided job. The result will be sent back to master by using DICKPTs. The end of the parallel region, the master sends the result of the checkpoint to every slaves to synchronize the memory space of the program.

2.1.2 Transformations from OpenMP to CAPE source code. In CAPE framework, we defined and implemented a set of functions that perform tasks such as generate, distribute, receive, and inject DICKPTs into memory of program, etc.. Besides, in CAPE compiler, we define a set of prototypes to translate OpenMP program into the form of CAPE programs that can be executed into CAPE framework. At version 2, CAPE does not support nested and shared-data variables constructs. Therefore, CAPE just works for the OpenMP programs that match the Bernstein's conditions.

Using the prototypes in CAPE compiler, the OpenMP source code will be translated into CAPE form that does not contain OpenMP directives and structures any more. For more explanation about this mechanism, we will present a prototype for the *parallel for* construct, one of the most complex work-sharing constructs of OpenMP [5]. This will be shown in the Fig. 3.

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

↓ automatically translated into ↓

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

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

2.2 Checkpointing

Checkpointing is the technique that tries to save image of the state of processes at a point during its lifetime, and then, it can be resumed at the saving's time if necessary [19][4]. Processes can resume execution from a checkpoint state when a failure occurs, therefore, no need takes time to initialize and execute it from the beginning. These techniques have been introduced since two decades ago. Nowadays, they are researched and used widely on on fault-tolerance, applications of trace/debugging, roll-back/animated playback, and process migration.

In order to optimize the size of checkpoint, to reduce the time and overhead of checkpointing, there are many techniques which have been introduced. In those, incremental checkpoint [19][20][13][7][1] is the best technique. The main idea of incremental checkpoint is just saving the change of the state of program compared with the previous checkpoint.

For incremental checkpointing, detecting modified memory can be based on page-fault or hash-table mechanism. The granularity can be a page or a word. Besides, the researches applied some another techniques to reduce checkpoint size such as Memory exclusion [19], Checkpoint compression [19][20], User-directed checkpointing [19], combination of page protection and hash table [1], etc..

3 OPENMP DATA-SHARING ATTRIBUTE RULES

To synchronize the shared data from local memory to shared memory, OpenMP program has to determine data-sharing attribute for variables. The data-sharing attribute is identified when entering and exiting a `parallel` region and may be re-identified by entering and exiting location of other constructs that are placed inside the `parallel`. The identification rules are set either implicitly or explicitly.

3.1 Implicit rules

The implicit rules to determine the data-sharing attribute of variables are described in two groups: 1) variables referenced inside parallel region but outside any construct; 2) variables referenced in parallel region and in another construct. In this Section we only describe the rules applied in C/C++.

3.1.1 Data-sharing attribute rules for variables referenced inside parallel region but outside any construct. The OpenMP data-sharing attributes of variables that are referenced in `parallel` region, but not in a construct, are determined as follows:

- Variables declared outside `parallel` region are shared.
- Variables with static storage duration declared in the region are shared.
- Variables with const-qualified type having no mutable member are shared.
- Objects with dynamic storage duration are shared.
- Static data members are shared if they are not appeared in a `threadprivate` directive.
- Formal arguments of called routines in the region passed by reference inherit the data-sharing attributes of the associated actual argument.
- Other variables declared in the region are private.

3.1.2 Data-sharing attribute rules for variables referenced in parallel region and in another construct. In the another construct located inside `parallel`, the OpenMP data-sharing attribute of variables are inherited from the `parallel` region that belong to. In addition, it is re-determined implicitly by the rules as follows:

- Variables appearing in the `threadprivate` directive are private.
- The loop iteration variables in the `for` or `parallel for` are private.
- Object with dynamic storage duration is shared.
- Static data members are shared.
- Variables with static storage duration that are declared in a scope inside the construct are shared.

3.2 Explicit rules

In the OpenMP program, the data-sharing attribute of variables can be explicitly determined. To do that, it provides a `threadprivate` directive and a set of clauses associated with its constructs.

According to [18], `threadprivate` is a declarative directive. It specifies those variables replicated, in which each thread having its own copy. The syntax of the `threadprivate` directive is as follows:

```
#pragma omp threadprivate(list)
```

where `list` is a comma-separated list of file-scope, namespace-score, or static block-score variables that do not have incomplete types.

The syntax of OpenMP clauses is presented in Figure 4. They have to follow a construct that is able to accept theses clauses. Table 1 summaries what OpenMP data-sharing clauses are accepted by which OpenMP constructs.

```
#pragma omp directive-name [clause[ [,] clause]...]
      code-block
```

Figure 4: OpenMP directives syntax in `C/C++`.

	par-allel	for	sec-tions	sin-gle	parallel for	parallel sections
private	✓	✓	✓	✓	✓	✓
firstprivate	✓	✓	✓	✓	✓	✓
lastprivate		✓	✓		✓	✓
shared	✓				✓	✓
default	✓				✓	✓
copyin	✓				✓	✓
copyprivate				✓		
reduction	✓	✓	✓		✓	✓

Table 1: The summary of which OpenMP data-sharing clauses are accepted by which OpenMP constructs.

4 DATA-SHARING ON CAPE

Basing on distributed-memory mechanism, the memory is totally distributed on different nodes. Thanks this, CAPE can design and implement RC memory model as similar as OpenMP does. Here, all variables of program included shared and private variables are stored in the same behavior at all nodes executed the program. When reaching synchronization points, only the modified data of shared variables are extracted into TICKPT and synchronized between all nodes as illustration in Figure 5. In order to select and extract only modified data of shared variables, CAPE-TICKPT is designed within a set of rules and translated prototypes that implement implicit and explicit attribute rules for variables.

4.1 Implementing implicit rules

For detecting property of variables by default, CAPE is designed follow by the OpenMP rules with a small change. It is also added some functions to recognize the variables. There are the rules for adding functions and for determining the variables attributes implicitly:

(1) CAPE functions are added whenever the program meets the declarations of a variable. These functions save the address and attribute of the declared variables.

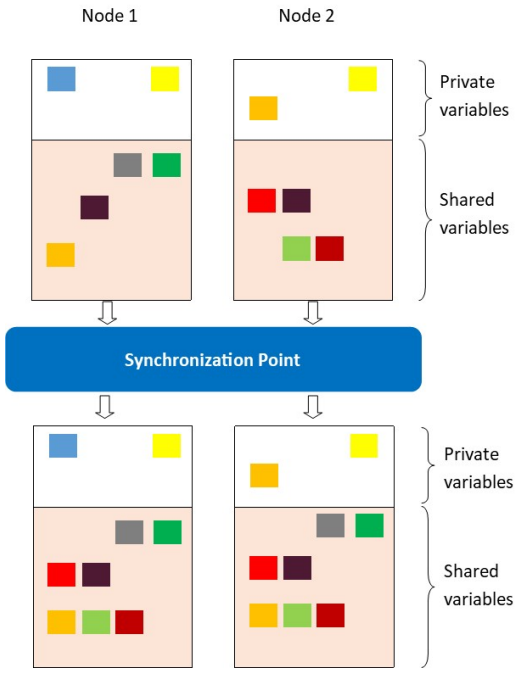

Figure 5: Updating of the shared variables between nodes.

(2) For dynamic variables, the allocation and destroying memory are handled by added CAPE functions. They add or remove these allocated addresses from variable list.
(3) The variables declared outside `parallel` region are shared, unless it is indicated in `threadprivate` directive.
(4) The loop iteration variables in the `for` or `parallel for` are changed from private on OpenMP to shared on CAPE in order to save the lastest iteration of this loop.
(5) All variables that are declared inside `parallel` construct and are not `static` or `dynamic` variables are private.
(6) Data-sharing attribute of variables of the constructs placed inside `parallel` construct are inherited from `parallel`.

4.2 Implementing explicit rules

For explicit rules of data-sharing attribute for variables, the OpenMP directive and clauses are translated into the relevant CAPE run-time functions in order to handle attributes. Figure 6 and Figure 7 present the forms that transform OpenMP `threadprivate` directive and clauses to CAPE code respectively. Here, the OpenMP `threadprivate` directive is translated to `cape_set_threadprivate()` function, and the OpenMP clauses are translated into relevant CAPE clause functions. The `cape_set_threadprivate()` function sets private attribute to the variable indicated by its address. The CAPE clause functions are implemented with the behaviors as well as the OpenMP clauses. They are responsible for changing of the variable attributes before taking checkpoints. The CAPE runtime functions associated with OpenMP clauses are presented and described in Table 2.

```
# pragma omp threadprivate(var1, var2, ...);
```

↓ is translated to ↓

```
cape_set_threadprivate(&var1);
cape_set_threadprivate(&var2);
...
```

Figure 6: Form to transform `pragma omp threadprivate` to CAPE function.

```
# pragma omp directive clause-1(var1)
                clause-2(var2) ...
        D ;
```

↓ is translated to ↓

```
begin-directive
        cape-clause-function-1(&var1, ...);
        cape-clause-function-2(&var2, ...);
        ...
        ckpt_start();
                D;
end-directive
```

Figure 7: Form to transform OpenMP constructs within data-sharing attribute clauses.

4.3 Generating and merging checkpoints

CAPE-TICKPT performs a series of operations to generate checkpoint that contains modified data of shared variables after executing a region of program. Two important steps of these operations are coping data at the beginning and finding different data at the end of region that marks checkpointing. Here, `ckpt_start()` function performs the first step. When called, it copies all data of variables that have `shared`, `lastprivate`, `copyin`, or `copyprivate` property (DATA-0). The second step is called at the synchronization point.

To find the different data, the CAPE monitor compares DATA-0 with current data (data that is read from memory at the time reaching synchronization point). The modified data is stored into checkpoint file. In addition, if `reduction` property is set to variables, the data of them is also saved into the checkpoint file regardless its changes. It is initial values and it is necessary to perform the indicated operator in `reduction` clause.

After generating checkpoints, they are sent it to partner nodes, these checkpoints are merged together using `mering` checkpoints operator as definitions and denotations bellows:

Definition 4.1. Checkpoint

A Checkpoint is both the registers values and the set of triples *address*, *value* and *timestamp* that represent the state of a running program at a time. It can be represented as follows:

$$C = (R_t, \{(a, v)_t\})$$

where:

- t is a timestamp, $t \in \{0, 1, 2, 3...N\}$.

OpenMP clause	CAPE runtime function	Description
default(none)	cape_set_default_none() ;	Sets property of all variables are `private`.
shared(a1, a2,...)	cape_set_shared(&a1); cape_set_shared(&a1); ...	Sets property of `a1, a2,...` variables are `shared`.
private(a1, a2, ...)	cape_set_private(&a1); cape_set_private(&a2); ...	Sets property of `a1, a2,...` variables are `private`.
firstprivate(a1, a2,...)	cape_set_firstprivate(&a1); cape_set_firstprivate(&a2); ...	Sets property of `a1, a2,...` variables are `firstprivate`.
lastprivate(a1, a2,...)	cape_set_lastprivate(&a1); cape_set_lastprivate(&a2); ...	Sets property of `a1, a2,...` variables are `lastprivate`.
copyin(a1, a2, ...)	cape_set_copyin(&a1); cape_set_copyin(&a2); ...	Sets property of `a1, a2,...` variables are `copyin`.
copyprivate(a1, a2, ...)	cape_set_copyprivate(&a1); cape_set_copyprivate(&a2); ...	Sets property of `a1, a2,...` variables are `copyprivate`.
reduction (Op: a1, a2, ...)	cape_reduction(&a1, OP); cape_reduction(&a2, OP); ...	Sets property of `a1, a2,...` variables are `reduction` with relevant operator.

Table 2: CAPE runtime functions associate with OpenMP clauses.

- R_t is the set of register values at time t and is called the register member of C. Denote $R_t = \{(r_i)_t\}, \forall r_i \in register\_values$.
- $\{(a, v)_t\}$ is the set of memory members of C, it saves value v at address a at time t, the memory member is saved word by word.

  Denote $S_t = \{(a, v)_t\} = \{(a_1, v_1)_{t_1}, (a_2, v_2)_{t_2}, ..., (a_n, v_n)_{t_n}\}$.

Definition 4.2. A `replacement` operation on checkpoints memory elements.

Let $(a, v_i)_{t_i}$ and $(a, v_j)_{t_j}$ be memory elements from different checkpoints having the same *address*. The `replacement` operation, denoted by $\odot$, is defined as follows:

$$(a, v_i)_{t_i} \odot (a, v_j)_{t_j} = (a, v_i \odot v_j)_{\max(t_i, t_j)} \quad (1)$$

where, $\odot$ can be any commutative and associative mathematical operations such as the OpenMP reduction operations if it is provided by program, otherwise, it is calculated by:

$$v_i \odot v_j = \begin{cases} v_i & \text{if } t_i > t_j \\ v_j & \text{if } t_i < t_j \end{cases} \quad (2)$$

Definition 4.3. `Merging` of Checkpoints memory members

The result of `Merging` two Checkpoint memory members $S_1 = \{(a_1, v_1)_{t_1}, (a_2, v_2)_{t_2}, \ldots, (a_i, v_i)_{t_i}\}$ and $S_2 = \{(a_1, v_1)_{t_1}, (a_2, v_2)_{t_2}, \ldots, (a_j, v_j)_{t_j}\}$ is a Checkpoint memory members, denoted $S_1 \oplus S_2$ that can be defined as follows:

$$\begin{aligned} S_1 \oplus S_2 = &\{(a_i, v_i)_{t_i} \in S_1 / \nexists (a_i, v_j)_{t_j} \in S_2\} \\ &\cup \{(a_j, v_j)_{t_j} \in S_2 / \nexists (a_j, v_i)_{t_i} \in S_1\} \\ &\cup \{(a_i, v_i)_{t_i} \odot (a_j, v_j)_{t_j} / \forall (a_i, v_i)_{t_i} \in S_1, \\ &\qquad \forall (a_j, v_j)_{t_j} \in S_2 \text{ and } a_i = a_j\} \end{aligned} \quad (3)$$

Definition 4.4. `Merging` of Checkpoints The result of `Merging` two checkpoints $C_1 = (R_{t_1}, S_1)$ and $C_2 = (R_{t_2}, S_2)$, denoted by $C = C_1 \oplus C_2$, is a checkpoint that can be represented as follows:

$$C = C_1 \oplus C_2 = (R_{\max(t_1, t_2)}, S_1 \oplus S_2) \quad (4)$$

The result of merging checkpoints after joining phase is injected into application's memory to update shared data.

5 ANALYSIS AND EVALUATION

To evaluate the new implementation, the matrix-matrix multiplication (MAMULT) and the program presented in Figure 8 (VECTOR) are conducted. It is designed basing on a Microbenchmark for OpenMP 2.0 [3][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;
}
```

Figure 8: The OpenMP function calculates on vectors using `for` construct.

The experiments are performed on a 16-node cluster with different computer's configurations. There are 2 computers with Intel(R) Pentium(R) Dual CPU E2160 @1.80GHz, 2GB of RAM, 5GB of free HDD; 7 computers with Intel(R) Core(TM)2 Duo CPU E7300 @2.66GHz, 3G of RAM, 6GB of free HDD; 5 computers with Intel(R) Core(TM) i3-2120 CPU @3.30GHz, 8GB of RAM, 6GB of free HDD; and 2 computers AMD Phenom(TM) II X4 925 Processor @2.80GHz, 2GB of RAM, 6GB of free HDD. All of these machines are operated the Ubuntu 14.03 LTS operation system, OpenSSH-Server and MPICH-2. They are joined into a LAN network with the capability 100Mbps of bandwidth.

To evaluate the performance of CAPE based on the new implementation, the execution time is measured and compare with CAPE previous version (CAPE-DICKPT) and MPI. Each experiment has been performed at least 10 times. The execution time is the mean of the items that match with 95% of confidence interval. Note that, CAPE-DICKPT has not supported multiple OpenMP directives, OpenMP clauses, and OpenMP programs that do not match with Bernstein's conditions yet. Therefore, only MAMULT program is used in this case.

Figure 9 and Figure 10 present the execution time in milliseconds of MAMULT program for various size of matrix and different sizes of cluster respectively. Note that, there are many kind of processors in different nodes, some of them are many cores, but a single core at each node was used during the experiments. Three are three measures that are presented at each time: the left one (yellow) is associated with CAPE-DICKPT (previous version), the middle one (blue) is associated with CAPE-TICKPT (current version), and the right one (red) is associated with MPI.

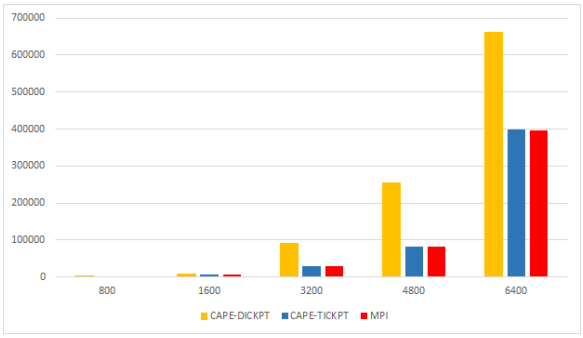

Figure 9: The execution time (in milliseconds) of MAMULT with different size of matrix on 16-node cluster.

Figure 9 presents the execution time for various of matrix sizes on 16-node cluster. The size is increased 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-DICKPT is much higher than that of CAPE-TICKPT and MPI (around 35%) while the execution time of CAPE-TICKPT and MPI are roughly equal.

Figure 10 presents the execution time for matrix size `6400x6400` on different size of the cluster. Number of nodes in turn is 4, 8, and 16. The result presented in this figure also shows the similar trend with the result on different matrix size. The execution time of MPI and CAPE-TICKPT is much lower than that of CAPE-DICKPT while the CAPE-TICKPT and MPI are nearly equal when the experiments are performed on 4-node, 8-node, and 16-node cluster. However,

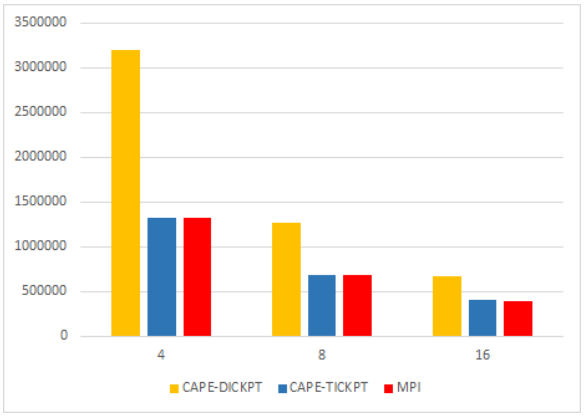

Figure 10: The execution time (in milliseconds) of MAMULT on different size of the cluster.

this figure also shows that the smaller the number of nodes is, the higher the difference in time taken when compared CAPE-DICKPT and CAPE-TICKPT or MPI is. It is easy to understand this because the number of nodes taking a part of jobs in CAPE-DICKPT is always less than 1 node compared with CAPE-TICKPT [23][24] and MPI. That means, on 4-node cluster, it is only 75% of the nodes involved in the computation phase in CAPE-DICKPT, while that in others are 100%. Therefore, in combination with the factors analyzed above, the execution time of CAPE-DICKPT on 4-node cluster is much higher than that of two other solutions. Although the difference in execution time decreases at the number of nodes that increases, CAPE-TICKPT always has a higher performance than CAPE-DICKPT.

The experiments above have demonstrated the performance of CAPE-TICKPT is higher than the previous version. Unfortunately, the previous version of CAPE has not supported to execute multiple OpenMP directives as well as OpenMP clauses in a program, so VECTOR program can not conducted to compare performance as well as checkpoint size of two versions. However, the reduction of checkpoint size can be demonstrated by implementation of DICKPT on the new execution model to compare checkpoint size of two methods.

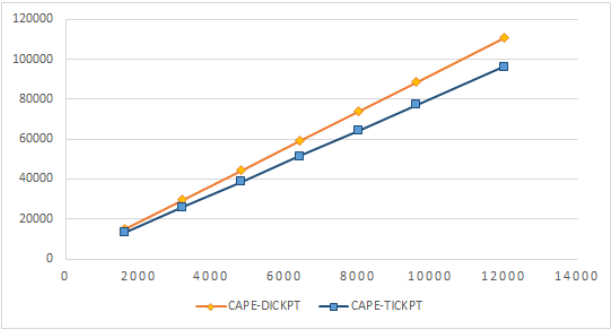

Figure 11: Checkpoint size (in bytes) after merging at the master node of two techniques.

Figure 11 shows the comparison of checkpoint size of two methods when executing VECTOR on 16-node cluster with various size of N and M. Here, we assigned $N = M$ and increased it from 2000 to 14000. According to the figure, the checkpoint size of CAPE-TICKPT is always smaller than that of CAPE-DICKPT in all measures. It is due to CAPE-TICKPT only extracts modified data of shared variables. That means, the modified data of variables i, vector A, and vector Z of VECTOR program are not saved into checkpoint file. Unlike CAPE-TICKPT, the previous method saves all modified data, that include variables i, j, A, B, Y, and Z.

6 CONCLUSION AND FUTURE WORKS

This paper presented implementation of OpenMP data-sharing attribute for variables on CAPE. It does not only provide a new capability of CAPE, but also contributes to reduce checkpoint size and improves the performance. The experiment shows that new method makes CAPE reduce the checkpoint size significantly. Moreover, we also found that the programmer can use CAPE efficiently if they use efficiently OpenMP clauses to manage data-sharing attribute for variables. We suggest using `default(none)` clause to set private to all variables, then using `share()` clause to set sharable property for variables that we want to share among processes.

REFERENCES

[1] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. 2004. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 277–286.

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

[3] J Mark Bull and Darragh O'Neill. 2001. A microbenchmark suite for OpenMP 2.0. *ACM SIGARCH Computer Architecture News* 29, 5 (2001), 41–48.

[4] Zhengyu Chen, Jianhua Sun, and Hao Chen. 2016. Optimizing Checkpoint Restart with Data Deduplication. *Scientific Programming* 2016 (2016).

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

[6] EPCC. [n. d.]. EPCC OpenMP micro-benchmark suite. ([n. d.]). https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite

[7] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. 2005. 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, 9.

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

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

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

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

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

[13] NAM Hyochang, KIM Jong, Sung Je Hong, and LEE Sunggu. 2002. Probabilistic checkpointing. *IEICE TRANSACTIONS on Information and Systems* 85, 7 (2002), 1093–1104.

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

[15] Laura Mereuta and Éric Renault. 2007. Checkpointing aided parallel execution model and analysis. In *High Performance Computing and Communications*.

Springer, 707–717.

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

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

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

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

[20] James S Plank, Jian Xu, and Robert HB Netzer. 1995. *Compressed differences: An algorithm for fast incremental checkpointing*. Technical Report. Citeseer.

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

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

[23] Van Long Tran, Éric Renault, Xuan Huyen Do, and Viet Hai Ha. 2017. Design and implementation of a new execution model for CAPE. In *Proceedings of the Eighth International Symposium on Information and Communication Technology (SoICT)*. ACM, 453–459.

[24] Van Long Tran, Éric Renault, Xuan Huyen Do, and Viet Hai Ha. 2018. A new execution model for improving performance and flexibility of CAPE. In *Parallel, Distributed and Network-Based Processing (PDP), 2018 26th Euromicro International Conference on*. IEEE, 234–238.

[25] Van Long Tran, Eric Renault, and Viet Hai Ha. 2016. Analysis and evaluation of the performance of CAPE. In *IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress. IEEE International Symposium on*. IEEE, 620–627.