Optimization of checkpointing and execution model for an implementation of OpenMP on distributed memory architectures

van Long Tran

DOCTORAT EN CO-ACCREDITATION
TÉLÉCOM SUDPARIS - INSTITUT MINES-TÉLÉCOM
ET L'UNIVERSITÉ PIERRE ET MARIE CURIE - PARIS 6

Spécialité: Informatique

École doctorale EDITE

Présentée par

Van Long TRAN

Optimization of checkpointing and Execution model for an implementation of OpenMP on Distributed Memory Architectures

Soutenue le 14/11/2018 devant le jury composé de:

Rapporteurs:	
Denis BARTHOU	Professeur - LaBRI - ENSEIRB-MATMECA - Bordeaux
Hacène FOUCHAL	Professeur - Université de Reims
Examinateurs:	
Christine MORIN	Directrice de recherche - IRISA - INRIA
Jean-Luc LAMOTTE	Professeur - Université Paris 6
Viet Hai HA	Docteur - Université de Hue - Viet Nam
Directeur de thèse :	
Éric RENAULT	Maître de conférences HDR - Télécom SudParis

Thèse numéro : 2018TELE0017

Acknowledgments

First of all, I would like to express my gratitude and thanks to my supervisor, Dr. Éric Renault, for his excellent guide, theoretical and technical help, useful discussion. He has encouraged and guided me very well both in research and in life in France with his great knowledge, patience, and understanding. I would like to thank for his financial support also.

I would like to thank Dr. Viet Hai Ha, my teacher from Hue University and my co-supervisor in this thesis for his encouragement, support, and helping during my life since I was went to university.

I deeply indebted to the Vietnamese government for the scholarship to study abroad. Thank College of Education, Hue University, Viet Nam and Hue Information of Technology Center (HueCIT), Viet Nam for their support to do experiments on their clusters. I also thank TSP for good working conditions.

I am grateful to Valérie Mateus, Xayplathi Lyfoung, Veronique Guy, and all members of TSP for their helps for administrative procedures.

I would like to thank Xuan Huyen Do and my friends living in France for their sharing and supports.

Finally, I deeply thank my wife, my daughter, and all of members in my family for their understanding, encouragement, and support. Thank my parents for helping us take care of my daughter since she was born. I dedicate this thesis to all the members of my family.

Abstract

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 the ability of OpenMP to automatically execute code in parallel and synchronize results using its directives, clauses, and runtime functions while MPI requires programmers do all this manually. Therefore, some efforts have been made to port OpenMP on distributed-memory architectures. However, excluding CAPE, no solution has successfully met both requirements: 1) to be fully compliant with the OpenMP standard and 2) high performance.

CAPE stands for Checkpointing-Aided Parallel Execution. It is a framework that automatically translates and provides runtime functions to execute OpenMP program on distributed-memory architectures based on checkpointing techniques. In order to execute an OpenMP program on distributed-memory system, CAPE uses a set of templates to translate OpenMP source code to CAPE source code, and then, the CAPE source code is compiled by a C/C++ compiler. This code can be executed on distributed-memory systems under the support of the CAPE framework. Basically, the idea of CAPE is the following: the program first run on a set of nodes on the system, each node being executed as a process. Whenever the program meets a parallel section, the master distributes the jobs to the slave processes by using a Discontinuous Incremental Checkpoint (DICKPT). After sending the checkpoints, the master waits for the returned results from the slaves. The next step on the master is the reception and merging of the resulting checkpoints before injecting them into the memory. For slave nodes, they receive different checkpoints, and then, they inject it into their memory to compute the divided job. The result is sent back to the master using DICKPTs. At 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 as a whole.

In some experiments, CAPE has shown very high-performance on distributed-memory systems and is a viable and fully compatible with OpenMP solution. However, CAPE is in the development stage. Its checkpoint mechanism and execution model need to be optimized in order to improve the performance, ability, and reliability.

This thesis aims at presenting the approaches that were proposed to optimize and improve checkpoints, design and implement a new execution model, and improve the ability for CAPE. First, we proposed arithmetics on checkpoints, which aims at modeling checkpoint's data structure and its operations. This modeling contributes to optimize checkpoint size and reduces the time when merging, as well as improve checkpoints capability. Second, we developed TICKPT which stands for Time-stamp Incremental Checkpointing as an instance of arithmetics on checkpoints. TICKPT is an improvement of DICKPT. It adds a time-stamp to checkpoints to identify the checkpoints order. The analysis and experiments to compare it to DICKPT show that TICKPT do not only provide smaller in checkpoint size, but also has less impact on the performance of the program using checkpointing. Third, we designed and implemented a new execution model and new prototypes for CAPE based

on TICKPT. 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. Four, Open Data-sharing attributes are implemented on CAPE based on arithmetics on checkpoints and TICKPT. This also demonstrates the right direction that we took, and makes CAPE more complete.

List of Publications

[1] Van Long Tran, Éric Renault, and Viet Hai Ha. *Improving the Reliability and the Performance of CAPE by Using MPI for Data Exchange on Network.* In International Conference on Mobile, Secure and Programmable Networking (MSPN), pp. 90-100. Springer, 2015.

[2] Van Long Tran, Éric Renault, and Viet Hai Ha. *Analysis and evaluation of the performance of CAPE.* In 16th IEEE International Conference on Scalable Computing and Communications (ScalCom), pp. 620-627. IEEE, 2016.

[3] Viet Hai Ha, Xuan Huyen Do, Van Long Tran, and Éric Renault. *Creating an easy to use and high performance parallel platform on multi-cores networks.* In International Conference on Mobile, Secure and Programmable Networking (MSPN), pp. 197-207. Springer, 2016.

[4] Van Long Tran, Éric Renault, and Viet Hai Ha. *Optimization of checkpoints and execution model for an implementation of OpenMP on distributed memory architectures.* In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 711-714. IEEE Press, 2017.

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

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

[7] Van Long Tran, Éric Renault, Xuan Huyen Do, and Viet Hai Ha. *Time-stamp incremental checkpointing and its application for an optimization of execution model to improve performance of CAPE.* In International Journal of Computing and Informatics (Informatica), pp. 301-311, Vol 42, no. 3 , 2018.

[8] Van Long Tran, and Éric Renault. *Checkpointing Aided Parallel Execution: modèle d'exécution et optimisation.* Journées SUCCES, Grenoble 16-17 octobre 2017. (Poster)

[9] Van Long Tran, Éric Renault, Xuan Huyen Do, and Viet Hai Ha. *Implementation of OpenMP Data-Sharing on CAPE.* Ninth International Symposium on Information and Communication Technology (SoICT), Da Nang, Viet Nam, 6-7/12/2018. (accepted)

[10] Van Long Tran, Éric Renault, and Viet Hai Ha. *Arithmetic on checkpoints and application for optimization checkpoints on CAPE.* (to be submitted)

[11] Van Long Tran, Éric Renault, Viet Hai Ha, and Xuan Huyen Do. *Using TICKPT to improve performance and ability for CAPE.* (to be submitted)

Résumé

OpenMP et MPI sont devenus les outils standards pour développer des programmes parallèles sur une architecture à mémoire partagée et à mémoire distribuée respectivement. Comparé à MPI, OpenMP est plus facile à utiliser. Ceci est dû au fait qu'OpenMP génère automatiquement le code parallèle et synchronise les résultats à l'aide de directives, clauses et fonctions d'exécution, tandis que MPI exige que les programmeurs fassent ce travail manuellement. Par conséquent, des efforts ont été faits pour porter OpenMP sur les architectures à mémoire distribuée. Cependant, à l'exclusion de CAPE, aucune solution ne satisfait les deux exigences suivantes: 1) être totalement conforme à la norme OpenMP et 2) être hautement performant.

CAPE signifie «Checkpointing-Aided Parallel Execution». C'est un *framework* qui traduit et fournit automatiquement des fonctions d'exécution pour exécuter un programme OpenMP sur une architecture à mémoire distribuée basé sur des techniques de checkpoint. Afin d'exécuter un programme OpenMP sur un système à mémoire distribuée, CAPE utilise un ensemble de modèles pour traduire le code source OpenMP en code source CAPE, puis le code source CAPE est compilé par un compilateur C/C++ classique. Fondamentalement, l'idée de CAPE est que le programme s'exécute d'abord sur un ensemble de nœuds du système, chaque nœud fonctionnant comme un processus. Chaque fois que le programme rencontre une section parallèle, le maître distribue les tâches aux processus esclaves en utilisant des checkpoints incrémentaux discontinus (DICKPT). Après l'envoi des checkpoints, le maître attend les résultats renvoyés par les esclaves. L'étape suivante au niveau du maître master consiste à recevoir et à fusionner le résultat des checkpoints avant de les injecter dans sa mémoire. Les nœuds esclaves quant à ceux reçoivent les différents checkpoints, puis l'injectent dans leur mémoire pour effectuer le travail assigué. Le résultat est ensuite renvoyé au master en utilisant DICKPT. À la fin de la région parallèle, le maître envoie le résultat du checkpoint à chaque esclave pour synchroniser l'espace mémoire du programme.

Dans certaines expériences, CAPE a montré des performances élevées sur les systèmes à mémoire distribuée et constitue une solution viable entièrement compatible avec OpenMP. Cependant, CAPE reste en phase de développement, ses checkpoints et son modèle d'exécution devant être optimisés pour améliorer les performances, les capacités et la fiabilité.

Cette thèse vise à présenter les approches proposées pour optimiser et améliorer la capacité des checkpoints, concevoir et mettre en œuvre un nouveau modèle d'exécution, et améliorer la capacité de CAPE. Tout d'abord, nous avons proposé une arithmétique sur les checkpoints qui modélise la structure des données des checkpoints et ses opérations. Cette modélisation contribue à optimiser la taille des checkpoints et à réduire le temps nécessaire à la fusion, tout en améliorant la capacité des checkpoints. Deuxièmement, nous avons développé TICKPT qui signifie «Time-Stamp Incremental Checkpointing» une implémentation de l'arithmétique sur les checkpoints. TICKPT est une amélioration de DICKPT, il a ajouté l'horodatage aux checkpoints pour identifier l'ordre des checkpoints. L'analyse et les expériences comparées à DICKPT montrent TICKPT sont non seulement plus petites, mais qui'ils ont également moins d'impact sur les performances du programme. Troisièmement, nous avons conçu et implémenté un nouveau modèle d'exécution et de nouveaux prototypes pour CAPE basés sur TICKPT. Le nouveau modèle d'exécution permet à CAPE d'utiliser les ressources efficacement, d'éviter les risques de goulots d'étranglement et de satisfaire à l'exigence des les conditions de Bernstein. Au final, ces approches améliore significativement

les performances de CAPE, ses capacités et sa fiabilité. Le partage des données implémenté sur CAPE et basé sur l'arithmétique sur des checkpoints est ouvert et basé sur TICKPT. Cela démontre également la bonne direction que nous avons prise et rend CAPE plus complet.

Table of contents

List of Tables

List of Figures

Chapter 1

Introduction

Contents

1.1 Introduction

In order to minimize programmers' difficulties when developing parallel applications, a parallel programming tool at a higher level should be as easy-to-use as possible. MPI [1], which stands for Message Passing Interface, and OpenMP [2] are two widely-used tools that meet this requirement. MPI is a tool for high-performance computing on distributed-memory environments, while OpenMP has been developed for shared-memory architectures. If MPI is quite difficult to use, especially for non programmers, OpenMP is very easy to use, requesting the programmer to tag the pieces of code to be executed in parallel.

Some efforts have been made to port OpenMP on distributed-memory architectures. However, apart from Checkpointing-aided Parallel Execution (CAPE) [3, 4, 5], no solution successfully met the two following requirements: 1) to be fully compliant with the OpenMP standard and 2) high performance. Most prominent approaches include the use of an SSI [6], SCASH [7], the use of the RC model [8], performing a source-to-source translation to a tool like MPI [9, 10] or Global Array [11], or Cluster OpenMP [12].

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 [6] have shown that the larger the number of threads, the lower the performance. As a result, in order to reduce the execution time overhead involved by the use of an SSI, other approaches have been proposed. For example, SCASH that maps only the shared variables of the processes onto a

shared-memory area attached to each process, the other variables being stored in a private memory, and the RC model that uses the relaxed consistency memory model. However, these approaches have difficulties to identify the shared variables automatically. As a result, no fully-compliant implementation of OpenMP based on these approaches has been released so far. Some other approaches aim at performing a source-to-source translation of the OpenMP code into an MPI code. This approach allows the generation of high-performance codes on distributed-memory architectures. However, not all OpenMP directives and constructs can be implemented. As yet another alternative, Cluster OpenMP, proposed by Intel, also requires the use of additional directives of its own (ie. not included in the OpenMP standard). Thus, this one cannot be considered as a fully-compliant implementation of the OpenMP standard either.

In order to bypass these limitations, CAPE is a solution that provides a set of prototypes and frameworks to automatically translate OpenMP programs for distributed memory architectures and make them ready for execution. The main idea of this solution is using incremental checkpointing techniques (ICKPT) [13, 14] to distribute the parallel jobs and their data to the other processes (the fork phase of OpenMP), and collect the results after the execution of the jobs from all processors (the join phase of OpenMP).

CAPE is developed over two stages. In the first stage of development, CAPE used Complete Checkpointing [3]. The checkpointing extracts all modification of data and stores in checkpoint files before sending them to slave node to distribute jobs. By this way, the result checkpoints are also extracted from slave nodes and sent back to the master to merge together. This approach has demonstrated the fully compliant with OpenMP of CAPE, but the performance is not so good. In the second stage, CAPE used Discontinuous Incremental Checkpointing [4, 15, 5] – a kind of Incremental Checkpointing. The checkpointing extracts only the modified data of the master nodes when dividing jobs and sends them to slave nodes to distributed jobs. After execution the divided jobs, each slave node also extracts the modified data and stores into a checkpoint, then sends back to the master node to merge and inject into its application memory. This approach has shown that CAPE is not only fully compliant with OpenMP, but also improve significantly the performance.

Although CAPE is still under development, it has shown its ability to provide a very efficient solution. For instance, a comparison with MPI showed that CAPE is able to reach up to 90% of the MPI performance [4, 15, 5]. This has to be balanced with the fact that CAPE for OpenMP requires the introduction of few `pragma` directives only in the sequential code, ie. no complex code from the user point of view, while writing an MPI code might require the user to completely refactorise the code. Moreover, as compared to other OpenMP for distributed-memory solutions, CAPE promises a fully compatibility with OpenMP [4, 5].

Besides the advantages, CAPE should be continued to develop more optimally. The main restrictions in this version are:

- CAPE can only execute OpenMP programs that match the Bernstein's conditions, only OpenMP `parallel for` directives with no clauses and nested constructs have been implemented in the current version.

- Discontinuous Incremental Checkpointing (DICKPT) as applied on CAPE extracts all modified data including private variables. Therefore, it contains unnecessary data in checkpoints. This leads to a larger amount of the data transferred over the network and reduces the reliability of CAPE.

- Checkpoints on CAPE do not contain any information that identifies the order of the generated checkpoints on the different nodes. Therefore, the merging of checkpoints requires that no memory elements appear in different checkpoints at the same address, or the checkpoints need to be ordered.

- For the execution model, CAPE always takes 1 node to be responsible for the master node. It divides jobs, distributes them to slave nodes, and waits for results but does not take any part in the execution of the divided jobs. This mechanism not only wastes the resources but also leads to risk of bottleneck. The bottleneck may occur at the join phase when checkpoints from all slave nodes are sent to the master node, as the master receives, merges the checkpoints all together, and sends the result back to all the slave nodes.

1.2 Problem definition

After analyzing the restrictions, this thesis focuses on the optimization of checkpoints and the execution model of CAPE. This improves the performance, reliability, and ability of this solution.

For the optimization of checkpoints, we focus on modeling the checkpointing techniques and applying on CAPE to improve CAPE's ability, reliability as well as performance. Here, the data structures of checkpoints need to be modeled to provide operations that are used directly to compute on checkpoints. Therefore, it reduces the amount of duplicated data in a checkpoint and the execution time when merging checkpoints. Based on this model, a new checkpointing technique has been developed and applied on CAPE to solve the three first restrictions indicated above.

For the optimization of the execution model, we designed and implemented a new execution model based on this new checkpointing technique. In this model, all nodes are involved in the computation of a part of the divided jobs to reduce the execution time. In addition, the new merging mechanism takes benefits of the new checkpointing techniques and is applied to avoid the risk of bottleneck as well as contributing to improve the perform of CAPE.

1.3 Organization of the thesis

The thesis is organized as follows: Chap. 2 presents the state of the art. The solutions for the optimization of checkpoints and the execution model of CAPE are presented in Chap. 3 and 4. Chapter 5 describes the methods to implement OpenMP data-sharing on CAPE. The conclusion and future works of this thesis are presented in Chap. 6.

In Chap. 2, the two main standard tools in parallel programming, i.e. OpenMP and MPI, are reminded. Then, we present an overview of the solutions that tried to port OpenMP on distributed-memory architectures, and together with checkpoint techniques. In addition, the previous version of CAPE is summarized and presented in this chapter.

Chapter 3 presents the modeling on checkpoints including checkpoint definitions and operations. Based on this model, Time-stamp Incremental CheckPoinT (TICKPT) is introduced as a case study. Then, Sec. 3.4 presents the analysis and evaluation of these approaches and compares them with DICKPT.

Chapter 4 presents a new design and implementation for an execution model of CAPE based on TICKPT. Section 4.2 and 4.3 describe the abstract model and the implementation of the CAPE's memory model respectively. Section 4.4 presents a new execution model for CAPE while Sec. 4.5 presents CAPE's transformation prototypes. The analysis and performance evaluation are presented in Sec. 4.6

Chapter 5 describes the implementation of the OpenMP Data-sharing model on CAPE and reminds OpenMP data-sharing attribute rules. Then, Sec. 5.3 and 5.4 present the implementation as well as the analysis and evaluation of this approach on CAPE.

The last chapter highlights the contributions of this thesis and some points to continue to develop in the future.

Chapter 2

State of the Art

Contents

2.1 OpenMP and MPI

OpenMP [2] and MPI [1] are two widely used tools for parallel programming. Originally, they run on shared and distributed memory architecture systems respectively.

2.1.1 OpenMP

OpenMP provides a high level of abstraction for the development of parallel programs. It consists of a set of directives, functions and environment variables to easily support the transformation of a C, C++ or Fortran sequential program into a parallel program. A programmer can start writing a program on the basis of one of the supported languages, compile it, test it, and then gradually introduce parallelism by the mean of OpenMP directives.

2.1.1.1 Execution model

OpenMP follows the fork-join execution model with threads as presented in Figure 2.7. When the program starts, only one thread is running (the master thread) and it executes the code sequentially. When it reaches an OpenMP parallel directive, the master thread spawns a team work including itself and a set of secondary threads (the fork phase), and the work allocated to each thread in this team starts. After secondary threads complete their allocated work, results are updated in the memory space of the master thread and all secondary threads are suspended (the join phase). Thus, after the join phase, the program executes only one thread as per the original. Fork-join phases can be carried out several times in a program and can be nested.

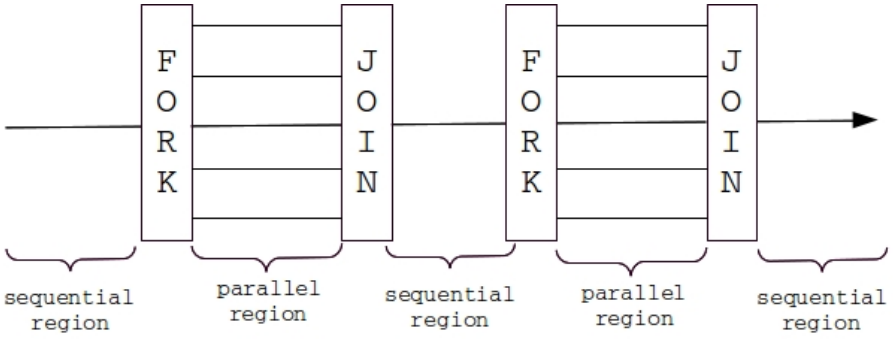

Figure 2.1: The Fork-Join model.

2.1.1.2 Memory model

OpenMP is based on the use of threads. Thus, it can only run on shared-memory architectures as well as all threads use the same memory area. However, in order to speed up the computations, OpenMP uses the relaxed-consistency memory model [16]. With this memory model, threads can use a local memory to improve memory accesses. The synchronization of the memory space of the threads is performed automatically both at the beginning and at the end of each parallel construct, or can be performed manually by specifying flush directives. Figure 2.2 illustrates shared and local memory in an OpenMP program.

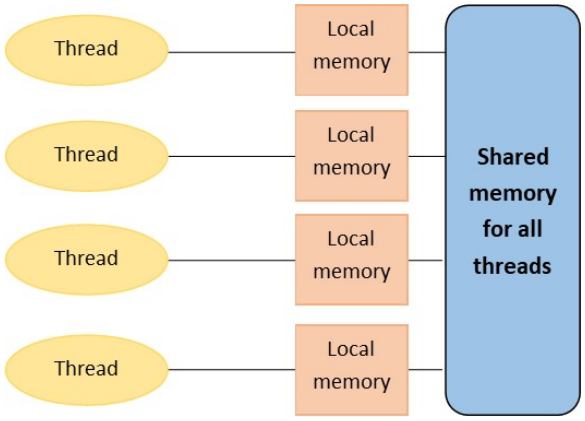

Figure 2.2: Shared and local memory in OpenMP.

2.1.1.3 Directives, clauses, and functions

For C, C++ or Fortran compilers supporting OpenMP, directives and clauses are taken into consideration through the execution of specific codes. OpenMP offers directives for the parallelization of most classical parallel operations including parallel region, worksharing, synchronization, data environment, etc. In addition, OpenMP provides a set of functions and environment variables in order to set or get the value of runtime environment variables.

Figure 2.3 presents the syntax of OpenMP directives in `C/C++`. Each directive has to start with `#pragma omp` followed by the name of the directive. Then, depending on the properties of the directive, some clauses might be provided to monitor the synchronization or data sharing properties.

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

Figure 2.3: OpenMP directives syntax in `C/C++`.

2.1.1.4 Example

Figure 2.4 presents an example of translation of a piece of sequential code into a parallel code using OpenMP. The sequential block is a function to compute the multiplication of two square

matrices, and can easily be transformed into a parallel code based on OpenMP directive. The parallel code can execute in parallel under support of the OpenMP environment.

```
void matrix_mult(int A[], int B[], int C[], int n){
        int i, j, k;
        for(i = 0; i < n; i ++)
                for(j = 0; j < n; j++ )
                        for ( k = 0; k < n; k++)
                                C[i][j] += A[i][k] * B[k][j];
}
```

↓ is easy to translate to parallel code ↓

```
void matrix_mult(int A[], int B[], int C[], int n){
        int i, j, k;
        #pragma omp parallel for
        for(i = 0; i < n; i ++)
                for(j = 0; j < n; j++ )
                        for ( k = 0; k < n; k++)
                                C[i][j] += A[i][k] * B[k][j];
}
```

Figure 2.4: Translation from sequential code to parallel code using OpenMP directives.

2.1.2 MPI

MPI is an Application Programming Interface (API) developed during the 90s. This interface provides essential point-to-point communications, collective operations, synchronization, virtual topologies, and other communication facilities for a set of processes in a language-independent way, with a language-specific syntax, plus a small set of language-specific features.

MPI uses the SPMD (Single Program Multiple Data) programming model where a single program is run independently on each node. A starter program is used to launch all processes, with one or more processes on each node and all processes first synchronize before executing any instructions. At the end of the execution, before exiting, processes all synchronize again. Any exchange of information can occur in between these two synchronizations. A typical example is the master-slave paradigm where the master distributes the computations among the slaves and each slave returns its result to the master after the job completes.

Although it is capable of providing high performance and running on both distributed-memory architectures, MPI programs require programmers to explicitly divide the program into blocks. Moreover, some tasks, like sending and receiving data or the synchronization of processes, must be explicitly specified in the program. This makes it difficult for programmers for two main reasons. First, it requires the programmer to organize the program into parallel structures which are bit more complex structures than in sequential programs. This is even more difficult for the parallelization of a sequential program, as it severely disrupts

the original structure of the program. Second, some MPI functions are difficult to understand and use. For these reasons, and despite the fact that MPI achieves high performance and provides a good abstraction of communications, MPI is still not very easy to use for parallel programming.

Figure 2.5 illustrates a piece of code that parallelize the sequential code presented in Section 2.1.1.4. It is important to note that the division of tasks for processors and synchronization of data between processes are explicitly done by programmers. The programmers can do in different algorithms, and receive different performances.

```
void matrix_mult(int A[], int B[], int C[], int n){
       int numtasks, taskid, i, j, k;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  //divide tasks for the processors
  int __l__ = taskid * n /numtasks ;
  int __r__ = (taskid + 1) * n / numtasks ;
  //caculate at each processor
  for(i = __l__; i < __r__; i ++)
       for(j = 0; j < N; j++ )
                              for ( k = 0; k < n; k++)
                                     C[i][j] += A[i][k] * B[k][j];

  //syncronize the result
  int nsteps = log2 (numtasks);
  unsigned long msg_size = (n * n )/ numtasks ;
  int partner;
  int __nl__, __nr__;
  for(i = 0; i < nsteps; i ++){
              partner = taskid ^ (1 << i);
              MPI_Sendrecv(&__l__, 1, MPI_INT, partner, i,
                                    &__nl__, 1, MPI_INT, partner, i,
                                    MPI_COMM_WORLD, &status);
              MPI_Sendrecv(&C[__l__][0], msg_size, MPI_INT, partner, i,
                                    &C[__nl__][0], msg_size, MPI_INT,
                                        partner, i,
                                    MPI_COMM_WORLD, &status);
              __l__ = (__l__ <= __nl__) ? __l__ : __nl__ ;
              msg_size = msg_size * 2 ;
  }
  MPI_Finalize();
}
```

Figure 2.5: Translation of the sequential code into MPI code by programmers.

2.2 OpenMP on distributed memory architectures

Many efforts have tried to port OpenMP on distributed memory architecture system to provide a very easy-to-use tool at higher level. The typical approaches are listed below.

2.2.1 Solutions based on translation to software DSM

Distributed Shared Memory system (DSM) is the traditional approach to allow OpenMP on clusters. The goal of this method is to transparently implement shared-memory paradigm on a distributed-memory system without any modifications of the source code. Some software DSMs have their own API, so that a translator has been provided to translate shared memory paradigms of OpenMP into relevant calls to these API. Some of them were designed to implement OpenMP without source code translation. As well as OpenMP memory model, to improve performance, the relaxed consistency model is used to implement shared memory abstraction in software DSMs (e.g. TreadMark [17] and SCASH [7]). However, in some approaches, a restricted sequential consistency model is used to increase the portability of the implementation (e.g. NanosDSM [18]).

The main issues of performance in software DSMs are the need to maintain coherence of the shared memory and global synchronizations at the barriers. Therefore, most software DSMs relax coherent semantics and impose a size limitation on the shared area to reduce the cost of coherent maintenance. A memory management module is used to detect accesses to shared memory, usually by page granularity. In the case of reading and writing to the same page by different processors, the writing processor invalidates all copies of the page more than the one in its local cache, which results in a page fault when the reading processor tries to use it. That is called *false sharing* problem. In some software DSMs, such as TreadMarks or SCASH, the multiple writer protocol [17, 7, 19] allows different processors to write a page at the same time and merges their modifications at the next synchronization point, so that the *false sharing* problem can be reduced in some context. In another efforts, the *false sharing* can also be reduced by using cache-line sized granularity [20] and single-byte granularity [21].

As can be seen in the results shown in [19], a native translation of a realistic shared-memory parallel program for distributed-memory systems using software DSMs cannot provide satisfactory performance.

2.2.1.1 TreadMarks

TreadMarks [17] provides an API for emulating shared memory, and the `SUIF` toolkit to translate OpenMP code to the code to be executed on the software DSM. Generally, the OpenMP program is translated to a Treadmarks program like that: OpenMP synchronization directives are replaced by TreadMarks synchronization operators; parallel regions are encapsulated into different functions and translated into a fork-join code; OpenMP's data environment is converted into shared data using parameters of procedures. In TreadMarks, shared and private variables are allocated on the heap and the stack respectively.

In TreadMarks, a multiple-writer protocol and a lazy invalidation version of Release Consistency (RC) is implemented to reduce the amount of communication involved in implementing the shared-memory abstraction. RC is a relaxed-memory consistency model that divides shared-memory accesses into acquire and release accesses, which are realized using

lock acquire and release operations. RC allows a process to write shared data in its local memory before it reaches a synchronization point.

To reduce the cost of a whole page movement, the concepts of *twin* and *diff* were introduced. The former copies a page when it is written for the first time, the later stores the modifications of a page by comparing the twin and current page.

2.2.1.2 Omni/SCASH

SCASH [22] is a page-based software DSM, based on the RC memory model with multiple writer protocols, so the shared memory area is synchronized at each synchronization point called barrier. It also provides `invalidate` and `update` protocols so that programmers can select one of them at runtime. `Invalidate` and `update` protocols send an invalidation message for a dirty page and the new data on a page to remote host where the page copy is kept at the synchronization point, respectively. For the shared memory areas, it has to be allocated explicitly by the shared memory allocation primitive at runtime, otherwise it is considered as a private area. This mechanism is called *shmem memory model*.

In SCASH, the concept of *base* and *home* node has been designed in order to implement the dynamic home node real location mechanism. The *home* node of a page is the node that stores the latest data of this page and the page directory that represents the nodes sharing the page. If a node wants to share a page, it has to copy the latest data of the page found on the *home* node. When reaching the *barrier*, the nodes having modified a shared page have to be sent to the *home* node the difference between the former and the new page data. The *home* updates the modified data into the page to maintain the latest page data. The *base* node is the node that stores the latest home node, and it has to be referenced by all nodes.

To execute in parallel on clusters, an OpenMP C/Fortran program can be translated to SCASH C/Fortran program [7, 23] using the Omni OpenMP compiler [24]. When SCASH is targeted, the compiler converts the OpenMP program into a parallel program using the SCASH static library, moving all shared variables into SCASH shared memory areas at runtime.

The big challenge of this approach is to translate OpenMP shared variables to the SCASH *shmem* memory model. In OpenMP, global variables are shared by default. However, in SCASH, variables declared in global scope remain private. To solve this, the translators use the following steps:

1. Transform all global variables to global variable pointers. These pointers point to a region in the shared address space where the actual global variables are stored.

2. Change the references of global variables to indirect references via the corresponding pointers.

3. Generate a global data initialization function for each compilation unit. These functions allocate the global variables in the shared address space and store their addresses in the corresponding global variable pointers.

2.2.1.3 ParADE

ParADE [25], which stands for Parallel Application Development Environment, is an OpenMP programming environment on top of a multi-threaded software DSM using a Home-based Lazy Release Consistency (HLRC) [26, 27] protocol. Figure 2.6 presents the architecture of

the ParADE programming environment. OpenMP translator and runtime system are the most important components of ParADE. The former translates the OpenMP source code into a ParADE code based on the ParADE API which in turn involves POSIX threads and MPI routines. The latter provides an environment to execute these APIs. To improve performance, the runtime system provides explicit message-passing primitives for synchronizing the small data structures at some synchronization points and work-sharing directives. This is a key feature of ParADE.

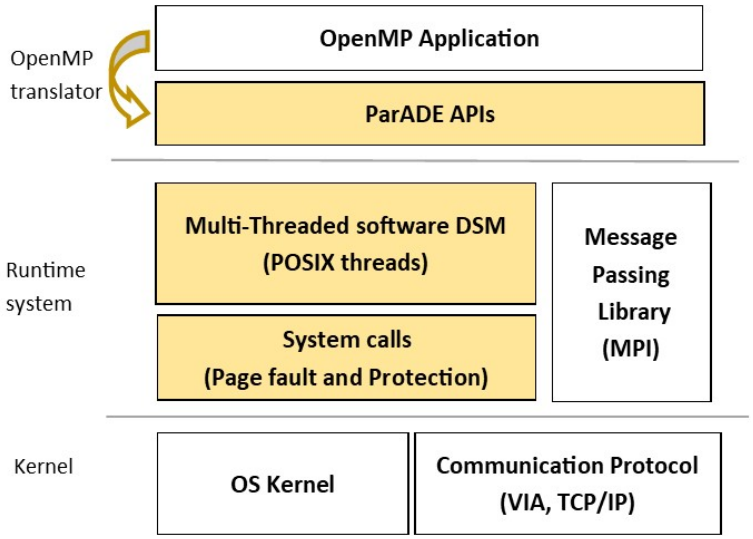

Figure 2.6: Architecture of the ParADE parallel programming environment.

In ParADE, OpenMP `parallel` directives are replaced by ParADE runtime interfaces to realize the fork-join execution model. The `for` work-sharing directive is only supported for `static` scheduling, and is recalculated the range of the iterations. The `critical` directive is translated to use `pthreadlock` and MPI to synchronize within a node and between nodes respectively.

Based on a multi-threaded software DSM that detects a process's access to shared memory pages by catching the SIGSEGV signal generated by the operating system, ParADE performs the operations to update the shared memory page to the remote nodes. ParADE also has to solve the atomic update and change right problem [28] or `mmap()` race condition [29]. This problem occurs when a thread in a node may access a shared page while another thread was handling a SIGSEGV signal and updating this page. To solve this, different methods have been introduced: using a file mapping, System V shared memory, a new `mdup()` system call, a `fork()` system call [25, 30].

2.2.1.4 NanosDSM

NanosDSM [18] is an everything-shared software DSM, which was implemented in Nthlib [31] runtime library designed to support distributed memory environment. The OpenMP program is compiled by NanosCompiler [32] to execute in parallel with NanosDSM on a DSM.

NanosDSM was designed achieving two conditions: 1) the whole address space is shared – not only the explicitly shared memory areas, but also stack and other regions are shared

among processes, 2) the system offers a sequential semantic – the modification at any node is updated immediately in the others.

To reach the sequential semantic, NanosDSM has implemented a single-writer multiple-readers coherence protocol [33]. Any node has to ask the permission to modify a memory page. The permission is managed by a master node of the page, and a page has a single master. This master node is migrated to any nodes depending on the first node that modified the page or the first node that started the application.

2.2.2 The use of a Single System Image

Single System Image (SSI) is the property of a system that presents users and applications a single unified computing resource [34]. It aims at focusing on a complete transparency of resource management, scalable performance, and system availability in supporting user applications on cluster-based systems. However, this mechanism is not enough to support the execution of OpenMP programs on cluster system directly as OpenMP threads are created and executed on a single machine.

Based on SSI, Kerrighed [35] is a single system image operating system for clusters which is successful to provide an environment to execute OpenMP programs on distributed-memory systems [6]. It supports memory sharing between threads and processes executing on different nodes of the cluster and aims at providing a high-performance, high-availability, and easy-to-use system [36]. In Kerrighed, the global shared memory is designed based on the container concept [37]. This container is a software object that allows storing and share memory with a page-based granularity on cluster.

The main advantage of Kerrieghed when executing OpenMP on cluster is its ability to provide a fully-compliant version of OpenMP. An OpenMP program can execute on this cluster system without any modifications. It is only a special compiler that needed linking target programs with the *gthreads* library. However, there are two factors that strongly reduce the global performance: the number of requirement of synchronizations and the physical expansion of the memory on the different machines [38]. In a demonstration [6], the results show that the larger the number of threads is, the lower the performance is (ref. Table 2.1).

Number of OpenMP threads	Execution time in seconds
1	19,78
2	31,57
4	42,71

Table 2.1: Execution time when running the HRM1D code on Kerrighed.

2.2.3 Solutions based on a Translation to MPI

MPI has become the standard programming tool for distributed-memory systems because of its high performance and portability. Specifying the pieces of code executing on each process and the explicit data communication mechanism between processes help to reduce the amount of transferred data over the network and make synchronization more flexible. However, it requires more efforts from the programmers. As a result, some efforts tried

to implement OpenMP on top of MPI [39, 9, 10], with the target to achieve have similar performance.

MPI uses a distributed-memory model where each process has its own memory space and no shared with the others. With OpenMP, the shared-memory and relax consistency memory models allow threads to use the same location in memory and have their own private memory. Therefore, to port OpenMP on MPI, the main challenge is to implement a shared-memory model on the distributed-memory system. There have been two different approaches to solve this problem [9, 10].

2.2.3.1 Extending OpenMP to directly generate communication code on top of MPI

In this approach, Antonio J. Dorta et al. introduced language `llc` – an extension of OpenMP, and a compiler `llCoMP` [10]. `llCoMP` is a source-to-source compiler implemented on top of MPI. It transforms OpenMP code with a combination of `llc` directives into `C` code which explicitly calls MPI routines. The resulting code is compiled into an executing code by a MPI compiler.

The `llc` language is designed basing on the OTOSP (One Thread is One Set of Processors) computational model [40]. The memory of each processor is private, the sequential codes are executed at the same in a set of processors – called a *processor set*. To implement the fork-join model, the set is divided into subsets (fork) as a consequence of the execution of a `parallel` directive, and they join back together (join) at the end of the execution of the directive. When joining, the modified data in the different subsets are exchanged.

For the translation of an OpenMP `parallel loop`, the shared variables in the left-hand side of an assignment statement inside a `parallel loop` have to be annotated with an `llc result` or `nc_result` clause. This notifies the compiler that these memory regions can be modified by the processor sets after being executed in the divided jobs.

In `llCoMP`, Memory Descriptors (MDE) are used to guarantee the consistency of the memory at the end of the execution of a parallel construct. MDE are composed of data structures that hold information in a set of pairs `(addr, size)`. They store the necessary information about memory regions modified by a processor set. The communication pattern involved in the transaction of a `llc result` or `nc_result` is *all-to-all* in most of the cases, and the data can be compressed to minimize the communication overhead.

This approach has shown the achievement of high performance in some experiments [10]. However, it also includes some drawbacks. It requires the programmer to add some pieces of code to identify the region of memory to be synchronized between processes. Moreover, other directives such as `single`, `master`, and `flush` are not easy to implement using the OTOSP model.

2.2.3.2 Translating OpenMP programs to MPI based on the partial replication model

In this approach, Ayon Basumallik et al. [9] use the concept of *partial replication* to reduce the amount of data communication between nodes. In this model, shared data are stored on all nodes, but no the shadow coping or management structure need to be stored. In addition, the arrays with fully-regular accesses are distributed between nodes by the compiler.

The fork-join model is transferred in to the Single-Program-Multiple-Data (SPMD) computational model [41] with the flowing characteristics:

- The `master` and `single` directives inside parallel regions, and the sequential regions are executed by all participating processes.
- Iterations of OpenMP parallel `for` loops are partitioned between processes using *block-scheduling*.
- Shared data are stored on all process memories.
- The concept of producers and consumers prevail for shared data and there is no concept like an owner for any shared data item.
- At synchronization points, each participating process exchanges the produced shared data.

Basically, translating from OpenMP to MPI follows these three steps:

1. *Interpretation of OpenMP Semantics Under Partial Replication.* In this step, the compiler converts the OpenMP program to the SPMD form. It translates the OpenMP directives, performs the requisite work partitioning, constructs the set of shared variables, and does computation re-partitioning. The shared variables are analyzed and stored in the shared memory areas of DSM using Inter-Procedural Analysis Algorithm for Recognizing Shared Data algorithm [42]. Computation re-partitioning transforms the patterns where the OpenMP application contains loops partitioned in the situation that multiple dimensional arrays are accessed along different subscripts in different loops. It then maintains data affinity by changing the level at which parallelism is specified and partitions work using the parallelism of the inner loops.
2. *Generation of MPI Messages using Array Dataflow.* In this step, the compiler inserts MPI functions to exchange data between producers and potential future consumers. The relationships between producers and consumers are identified based on performing a precise array-dataflow analysis. To characterize accesses to shared arrays, this compiler constructs bounded regular section descriptors [43]. A control flow graph is generated with a vertex mapped to a program statement. The regular section descriptors (RSDs) and stating points of OpenMP are recorded by annotating the vertices of the control flow graph and loop entry vertices, respectively. This graph is used to implement relaxed memory consistency model of OpenMP.
3. *Translation of Irregular Accesses.* Irregular accesses are the reading and the writing that cannot be analyzed precisely at compile-time. To handle that, a technique based on the property of *monotonicity* is used [9, 44]. However, in the cases that indirection functions are not provably monotonic or irregular write operation are not in the form of an *Array[indirection function]*, this compile-time scheme can not be applied. In these specific cases, a piece of code is inserted to record the write operation at runtime. These code allocated a buffer on each process to store the written elements in the form of an *(address, value)* pair. And this buffer is intercommunicated at the end of a loop that contains the irregular write operation.

The results presented in [9] have shown the high performance of this approach. However, only Steps 1 and 2 are implemented in the compiler. Step 3 has to be done manually. Therefore, this approach is also not fully compliant with OpenMP.

2.2.4 Intel Clusters OpenMP

Intel Cluster OpenMP (ClOMP) [12, 45] is the first commercial software which ports OpenMP on distributed-memory systems. It is developed by Intel corporation. It now fully supports OpenMP 2.5 standard for C, C++, and Fortran except for nested parallel regions.

ClOMP is implemented based on a software DSM. Moreover, to identify the shared variables among threads, it adds `shareable` directive into the OpenMP specification. This directive must be explicitly declared by the programmer in order to indicate the shared variables which are managed by the software DSMs.

In ClOMP, the runtime library is responsible for the consistency of shared variables across the distributed nodes. Shared variables are grouped together and allocated on certain pages in memory. The runtime library uses the `mprotect` system call to protect these pages against reading and writing. When the program reads from the page, the `SIGSEGV` signal is generated by the operation system, ClOMP executes a set of operations to update from all nodes that modified the page since it was last brought up-to-date. Then, the reading protection of the page is removed, the page is updated, and the instruction is restarted. When the program writes to a writing protection page, the similar page fault mechanism is performed, ClOMP makes a copy of the page (called a `twin`), and removes all protections from the page. The twin makes it possible to identify the modification in a page. At the synchronization points, the status of modification of these pages are sent to all nodes in the system to notify that they are not up-to-date.

Experiences in [12, 46] have shown the high performance of ClOMP. It takes advantage of the relaxed consistency memory model of OpenMP. However, users have to add `shareable` directive into some relevant pieces of codes to ensure it to perform correctly.

2.2.5 Using Global Array

Global Array (GA) [47, 48] is a set of library routines to simplify the programming methodology on distributed-memory systems. It aims at combining the better features of message passing and shared-memory – simple coding and efficient execution. The key concept of GA is providing a portable interface, where each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed matrices, and no need for explicit cooperation by other processes. That can be used by programmers to write parallel program on clusters using a shared memory access, specifying the layout of shared data at a higher level. In addition, the GA programming model acknowledges the difference of access time between remote and local memory, which requires the programmer to determine the needed locality for each phase of the computation. Furthermore, since GA is a library-based approach, the programming model works with the most popular language environments: currently bindings are available for FORTRAN, C, C++ and Python.

OpenMP programs can be automatically translated into GA programs because each has the concept of shared data and the GA library features match with most OpenMP directives. This statement has also been realized by the efforts of L. Hang et al. in [11, 49, 50, 51]. According to L. Hang et al., apart from the dynamic settings or changing of the number of threads like `omp_set_dynamic`, `omp_set_num_threads`, most OpenMP library routines and environment variables can also be translated into GA routines.

The strategy of translation from OpenMP to GA program as follows: a fixed number of processes are generated at the beginning of GA program, each OpenMP thread is mapped to a GA process. The *global arrays* are only shared variables in GA processes, the others are

private. OpenMP shared variables are translated to *global arrays* that are distributed and communicate automatically between processes. The most important transformations are:

- OpenMP `parallel` constructs are translated to `MPI_Init` and `GA_INITIALIZE` routines to initialize processes and the memory needed for storing distributed *global arrays* in GA.
- `GA_NODEID` and `GA_NNODES` are called to get ID and number of processes, respectively.
- OpenMP parallel loops (the loops inside parallel constructs) are translated into GA by invoking a generated GA program to calculate the loop bounds depending on the specified schedule so as to assign works. Depending on the new lower and upper bounds, and the array region accessed in the local code, the `GA_GET` in the GA program copy the relevant part of the *global arrays*. Then, the GA process performs its work and calls `GA_PUT` or `GA_ACCUMULATE` to put back the modified local copies into the global location.
- The `GA_CREATE` routine is called to distribute global arrays in the GA program whenever shared variables in parallel regions of OpenMP have appeared.
- The `GA_BRDCST` routine is responsible for the implementation of the OpenMP `firstprivate` and `copyin` clauses. The OpenMP `reduction` clause is translated by calling the `GA_DGOP` routine.
- OpenMP synchronizations are replaced by GA synchronization routines. As well as OpenMP, GA synchronizations ensure that all computations in parallel regions have been completed and their data has been updated to global arrays. For example, OpenMP `critical` and `automic` directives are translated by calling of GA locks and Mutex routine, GA `put` and `get` routines are used to replace OpenMP `flush`, `GA_SYNC` library calls are used to replace OpenMP `barrier` as well as implicit barriers at the end of OpenMP constructs, etc..
- The OpenMP `ordered` clause cannot be translated directly to GA routines, so `MPI_Send` and `MPI_Recv` are used to guarantee the execution order of processes.

In some experimental results [11, 49, 50, 51], the approach of translation from OpenMP to GA programs shown high performance. However, the drawbacks of this method are the requirements to explicitly specify shared variables, and some OpenMP routines that cannot be translated into GA routines. This problem prevents it to become a fully compliant with OpenMP on distributed-memory system.

2.2.6 Based on Checkpointing

CAPE [3, 4] is an approache based on checkpointing techniques [13, 52] to implement the OpenMP fork-join model. In that model, the program is only initialized in a thread called the master thread. The master thread is also the only one in charge of the execution of the sequential region. When it reaches the parallel region, the master thread creates additional threads – called slave threads, and distributes code and data to them automatically. That processing is called the `fork` phase. Then, all threads including the master execute the parallel code – called the `compute` phase. During the `compute` phase, data in each thread

is allowed to store locally. At the end of a parallel region – the `join` phase – data at all threads are updated into shared-memory, and all slave threads are suspended or released.

Two versions of CAPE have been developed so far. The first version (CAPE-1) [3] uses complete checkpoints, while the second version (CAPE-2) [4] uses incremental checkpoints. Both of two versions use processes instead of threads. Figure 2.7 shows the association of the OpenMP execution model and CAPE-1 execution model. The different phases are:

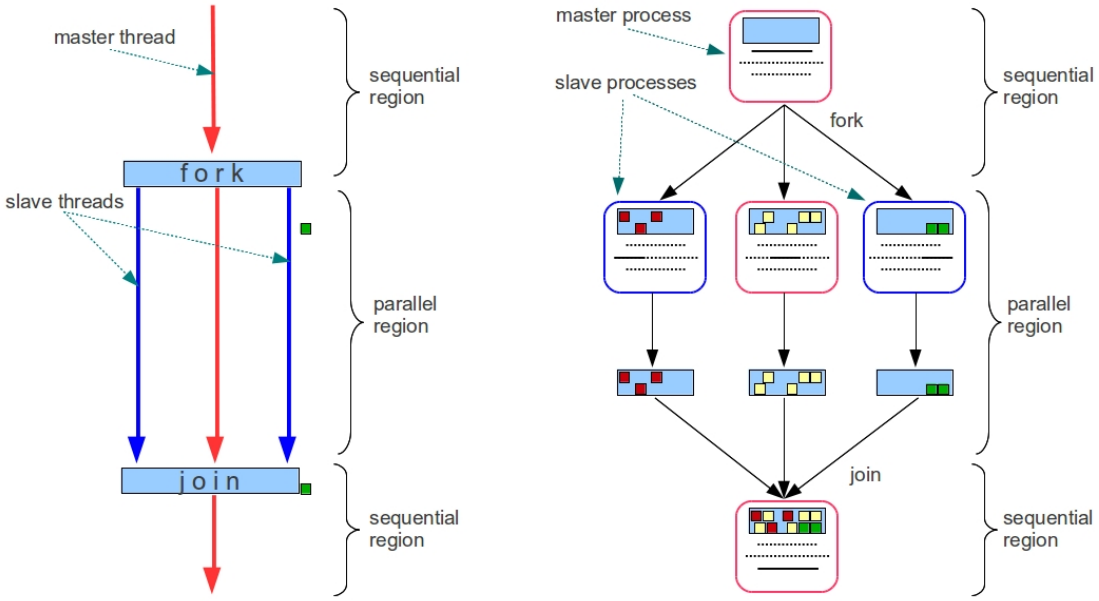

Figure 2.7: OpenMP execution model vs. CAPE execution model.

- `fork` *phase:* the master process divides the jobs into smaller parts or chunks and send them to slave processes. At the beginning of each part, the master process generates a complete checkpoint and sends it to a distant machine to create a slave process. After sending the last part, the master waits for the results from the slave processes.

- `compute` *phase:* the slave processes resume the process from the complete checkpoint sent by the master in order to compute the divided jobs. After computation, the result at each slave is extracted and saved into a complete result checkpoint. Unlike OpenMP, during this phase, the master process does not take part in the divided jobs. It only waits for the results from the slave processes.

- `join` *phase:* the result checkpoint at each slave process is sent to the master process, and all are merged altogether on the master. Then, the merged checkpoint is used to resume the execution. At this time, the master is the only active process in the system.

Currently, CAPE only supports for OpenMP program using C/C++. To execute on CAPE, the OpenMP program need to be translated to a CAPE program by a set of CAPE

prototypes. A CAPE program is a C/C++ program that substitutes OpenMP directives to CAPE functions that be compiled by any C/C++ compile: Figure 2.8 presents the translation of the OpenMP `parallel for` construct to a CAPE code with CAPE-1.

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

↓ automatically translated into ↓

```
1        parent = create ( original )
2        if ( ! parent )
3                exit
4        copy ( original, target )
5        for ( A ; B ; C )
6                parent = create ( before_i )
7                if ( parent )
8                        ssh host_x restart ( before_i )
9                else
10                       D
11                       parent = create ( after_i )
12                       if ( ! parent )
13                               exit
14                       diff ( before_i, after_i, delta_i )
15                       merge ( target, delta_i )
16                       exit
17        parent = create ( final )
18        if ( parent )
19                diff ( original, final, delta )
20                wait_for ( target )
21                merge ( target, delta )
22                restart ( target )
```

Figure 2.8: Template for OpenMP `parallel for` loops with complete checkpoints.

The CAPE-1 functions used in Fig. 2.8 are described as follows:

- `create (file)`: generates a complete checkpoint and save it into `file` .
- `copy (file`$_1$ `, file`$_2$`)`: copies the content of `file`$_1$ into `file`$_2$.
- `diff ( file`$_1$ `, file`$_2$ `, file`$_3$`)`: finds the difference between `file`$_1$ and `file`$_2$ and saves the result into `file`$_3$.
- `merge (file`$_1$ `, file`$_2$`)`: merges `file`$_1$ into `file`$_2$.
- `wait_for (file)`: waits for the results from slave processes and stores the resulting checkpoint in `file`.

- `restart (file)`: resumes the execution of the current process from the checkpoint file provided as a parameter.

In some comparisions [5, 53], CAPE has shown that it is a promising solution that can fully support OpenMP directives, clauses and functions. In addition, CAPE has achieved high performance. For more details, checkpointing techniques and CAPE-2 are presented in the next sections.

2.2.7 Summary

There have been many efforts to port OpenMP on distributed-memory systems. However, apart from Checkpointing-Aide Parallel Execution, there are no solution that are able to achieve the two requirements: 1) fully-compliant with OpenMP, and 2) high-performance. Table 2.2 presents the summary of advantages and drawbacks of all presented tools.

2.3 Checkpointing techniques

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 [13, 52]. 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. Figure 2.9 illustrates the data of a process that can be saved in a checkpoint file.

Basically, checkpointing are categorized [54, 55, 56, 57] into two approaches: system-level checkpointing (SLC) and application-level checkpointing (ALC).

System-level checkpointing is implemented and performed at the operation system level. It stores the whole state of the processes (register values, program counter, address space, etc.) in a stable storage. The main advantage of this method is its transparency. Programs can be checkpointed using SLC checkpointing without any efforts from the programmer or the user point of view. However it is no portable because it depends on the operating system library and system architectures. In addition, as compared with other approaches that only store necessary data, its cost is much higher. Some typical examples are Kerrighed [58] – checkpoint/recovery mechanisms for a DSM cluster system, CHPOX [59], Gobelins [60] – process migration as a primitive operation, and KeyKOS [61] – implementation of rollback recovery.

Application-level checkpointing is performed at the program level and only necessary data are stored into the checkpoint file. It can be transparent or non-transparent with the user. The ALC transparent methods [13, 62, 63, 64, 65] is achieved by compiling the program with a provided special checkpointing library, or by rewriting executable files, or injecting

Method	**Platform**	**Principle**	**Advantages**	**Drawbacks**
Tread-Marks [17]	DSM, cluster of SMPs	extends the original TreadMarks	high performance	difficulty to automatically identify shared variables
SCASH [7, 23]	DSM	maps only shared variables on the global shared memory	high performance	use of additional directives
RC model [25, 18, 21]	DSM	uses the HLRC model to implement the OpenMP memory model	high performance	difficulty to automatically identify shared variables and implementation of `flush`, `reduction`, `atomic`...
SSI [6, 35]	DSM	uses a SSI as a global memory for threads	fully OpenMP compliant	weak performance
llCoMP [10]	MPI	translates to MPI	high performance	use of additional directives; difficulty to implement `single`, `master`, `flush`
Partial Replication model [9]	MPI	translates to MPI; automatically specifies exchanged data between nodes	high performance	not completely implemented; difficulty to implement `single`, `master`...
ClOMP [12, 45]	DSM	maps only shared variables on the global shared memory	high performance	use of additional directives
GA [50, 11, 49, 51]	GA	translates to Global Array	high performance	difficulty to automatically identify shared variables
CKPT [3, 4]	Checkpoint	uses checkpoint techniques to implement the fork-join model	high performance and potential fully-compliant with OpenMP	current version for programs that only match Bernstein conditions

Table 2.2: Comparison of implementations for OpenMP on distributed systems.

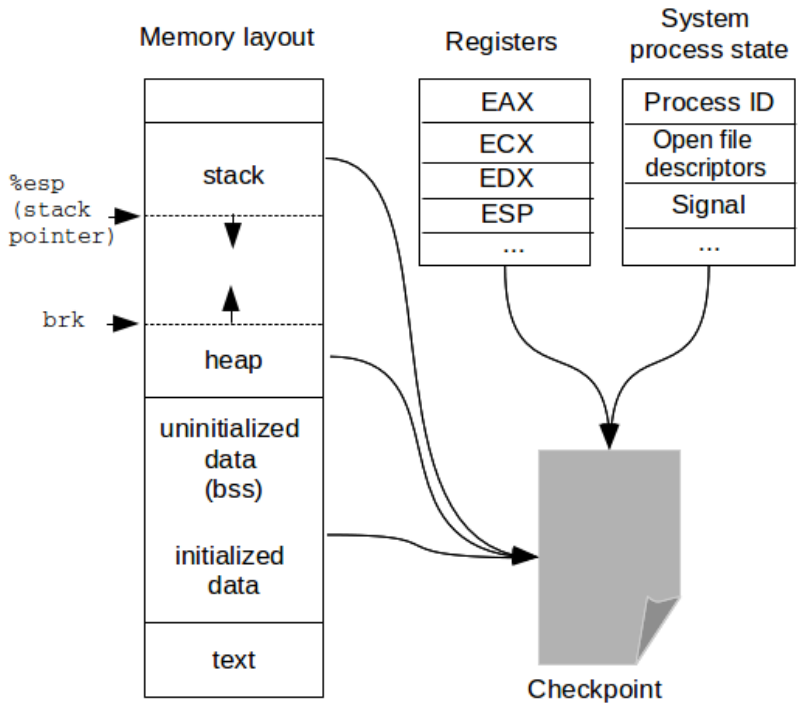

Figure 2.9: The state of a process can be saved in a checkpoint.

checkpointing into running processes. For ALC non-transparent methods [66, 67, 68], it requires adding checkpointing directives into the source code of the program by the user.

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

2.3.1 Complete checkpointing

Complete checkpointing [52, 69, 70] saves all information regarding the process at the points that it generates checkpoints. The advantages of this technique are reduction of the time of generation and restoration. However, not only a lot of duplicated data stored each time a checkpoint is taken, there are also duplications in the different generated checkpoints. For example, the code segment is never changed during execution but it is stored every time a checkpoint is taken. In the heap, few bytes of data maybe changed but it saves the whole segment. Therefore, checkpoints based on this techniques contain a lot of duplicated data, so that the checkpoint's size is too large. To solve this drawback, some methods have been proposed and applied:

- **Data Compression** is a straightforward approach to reduce checkpoint size based on data compression algorithms [71]. For example, the CATCH compiler [69] implemented on the LZW data compressor [72] that shows a significant reduction of the complete checkpoint size.

- **Data Deduplication** is a technique based on data compression to remove the repeating data [73, 74]. It splits the input data into different blocks and calculates a fingerprint for each of them. If there exists any blocks sharing the same fingerprint,

these blocks are considered as duplications, and checkpointing only need to save the index number for the duplicated blocks. Otherwise, it is unique, and data of that blocks has to be saved in the checkpoint file. The analysis and experiments in [70] shows a reducing factor from 10% to 30% of the checkpoint's size on a single node system, and from 47% to 50% on distributed memory systems.

- **Adaptive Checkpointing** aims at taking checkpoints at desirable points of time. It reduces checkpoint size by reducing the number of checkpoints. This method is useful for programs with large run-time variation in checkpoint size [69].

2.3.2 Incremental checkpointing

Incremental checkpointing [13, 75, 14, 76, 77, 55, 56, 78] 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.

The main challenges of these techniques is to detect the modified regions of the process's address space while it is running. Currently, four approaches are handling this problem:

- **Use of the `dirty` bit.** Figure 2.10 shows the structure of a Page Table Entry (PTE) in Linux Kernel for i386. The `dirty` bit is a high importance for memory management in Linux, especially for the page cache [55]. When a page is written, the hardware sets the `dirty` bit in the corresponding page. Therefore, when taking a checkpoint, all the PTEs are checked to identify if the page has been changed since the begin of the program or since the previous checkpoint. Then, all writable pages are marked as clean (status `non-dirty`). The biggest problem of this method is that all `dirty` bits are cleaned when the main memory associated with these pages written back from the cache. In order to maintain the correct kernel functionality, the original dirty bit is duplicated in one of the unused bits in the PTE [79]. The low-level functions used by the kernel to access this bit are properly updated.

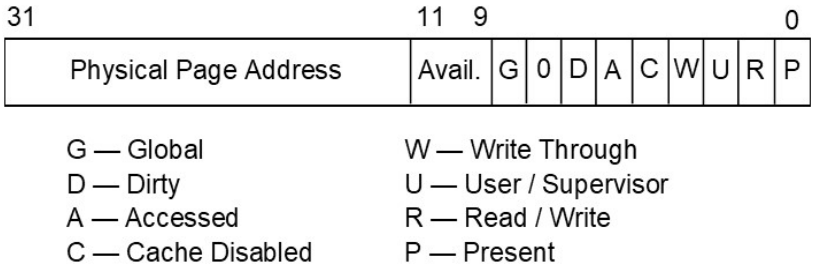

Figure 2.10: Page table entry in i386 version of the Linux Kernel.

- **Use of the virtual memory page fault handler.** After a checkpoint or at the beginning of the program, all writable pages of the virtual memory are set to write-protection. When the a page is written by the program for the first time, a page fault exception is thrown out. The page fault exception handler uses one of the available bits of the PTE (9th to 11th bit in Fig. 2.10) to mark the page has been modified, or saves the address of this page in a list. Then, the write-protection is removed, and the program can write in that page. When taking a checkpoint, the data of all modified pages is copied and stored into the checkpoint.

- **Use of a hash function.** This method consists in using a hash function to recognize the changes in the memory blocks [77]. In this method, a hash function $F()$ is selected to generate the hash values of the memory blocks. Each memory block X has a unique hash value $F(X)$. At the beginning of the program or after taking a checkpoint, the hash values of the memory blocks are computed and saved in a `hash table`. When taking the next checkpoint, the hash values of these blocks are re-computed and stored in another `hash table`. The two `hash tables` are compared to find the differences. If the hash values of a block in the two `hash tables` are different, this memory block is marked with a modified status, and is saved in the checkpoint file.

  There are two important factors in this methods: 1) the choice of the hash function, 2) identify the granularity of the blocks. For the hash function, theneeded space to store the hash values is much smaller than the size of the memory blocks, and it has to have correctness and performance, for instances, MD5, SHA1 and SHA2 [80]. Regarding the granularity, the smaller the block size is, the larger the opportunity to optimize the checkpoint size is. However, the performance maybe reduced because of taking time to compute the hash values. This granularity can be identified adaptively based on the variables in the program [81].

- **Use of variable-based approaches.** This method detects the modified variables and saves them into the checkpoint file. The detection of modified variables is presented in [82] where a compiler is modified to detect variable changes, and in [83] where an executable editing library is added.

For efficient optimization incremental checkpoints, beside applying optimizations on complete checkpoints, many other approaches have been proposed. Some typical approaches are 1) using live variable analysis [56] – ie. select variables are live during the checkpoint, avoiding storage of dead variables, 2) using user-directed checkpointing [13] – ie. exclude the unnecessary memory, or 3) using word-level granularity [76, 14] – ie. combine virtual memory page fault handler and save page's data to find the difference at word-level granularity, etc..

2.3.3 Discontinuous incremental checkpointing

Discontinuous Incremental Checkpointing (DICKPT) [14] was designed based on incremental checkpointing. It uses virtual memory page fault handler, combined with save page's data to detect the modified memory space at a word-level granularity. In addition, some additional directives are provided to improve the ability of taking checkpoint in a part of a program.

2.3.3.1 DICKPT mechanism

The mechanism of DICKPT is designed based on a virtual memory page fault handler on incremental checkpointing. However, it does not perform transparently. There are three `pragma` directives that are defined to identify the piece of code and the locations in the program taking checkpoints:

- `pragma dickpt start:` identifies the place that set write-protection to all writable pages and start monitoring.

- `pragma dickpt save file_name:` identifies the position that generates and saves incremental checkpoint into `file_name`.
- `pragma dickpt stop:` identifies the place that removes write-protection and stops monitoring.

When a modified page is identified by page fault handler, its data is copied into a list. At the time it takes a checkpoint, data in that list are compared to the current data in memory using word-level granularity. The different data in memory are stored in the checkpoint file.

2.3.3.2 Design and implementation

Figure 2.11 shows the abstract model of DICKPT on Linux Kernel for i386. It is implemented at both user level and kernel-levels [84].

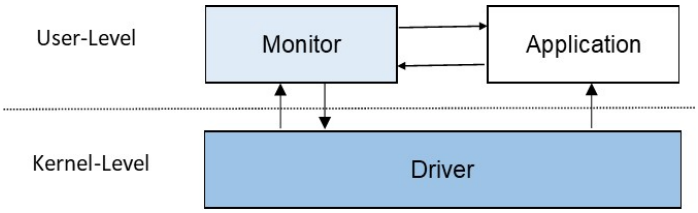

Figure 2.11: The abstract model of DICKPT on Linux.

Application: An application using DICKPT must be inserted the three `pragma` directives in to the pieces of code where the checkpoint needs to be taken. At execution, the application are called to execute and are traced by a monitor. Figure 2.12 presents a piece of code that illustrate the use of DICKPT's `pragma` directives in an applications.

```
int main ( int argc, char * argv [ ] ){
        functionA();
        #pragma dickpt start
        functionB();
        #pragma dickpt save file1.ckpt
        functionC();
        #pragma dickpt save file1.ckpt
        #pragma dickpt stop
        functionD();
        return 0 ;
}
```

Figure 2.12: An example of use of DICKPT's `pragma`.

When reaching `#pragma dickpt start`, `#pragma dickpt save`, and `#pragma dickpt stop` directives, signals `START`, `SAVE`, and `STOP` are respectively sent to the monitor. In addition, when the application writes to a memory page with the write protection set, the `SIGSEGV` signal is thrown out by the operating system. Then, this signal and the page address are also sent to the monitor.

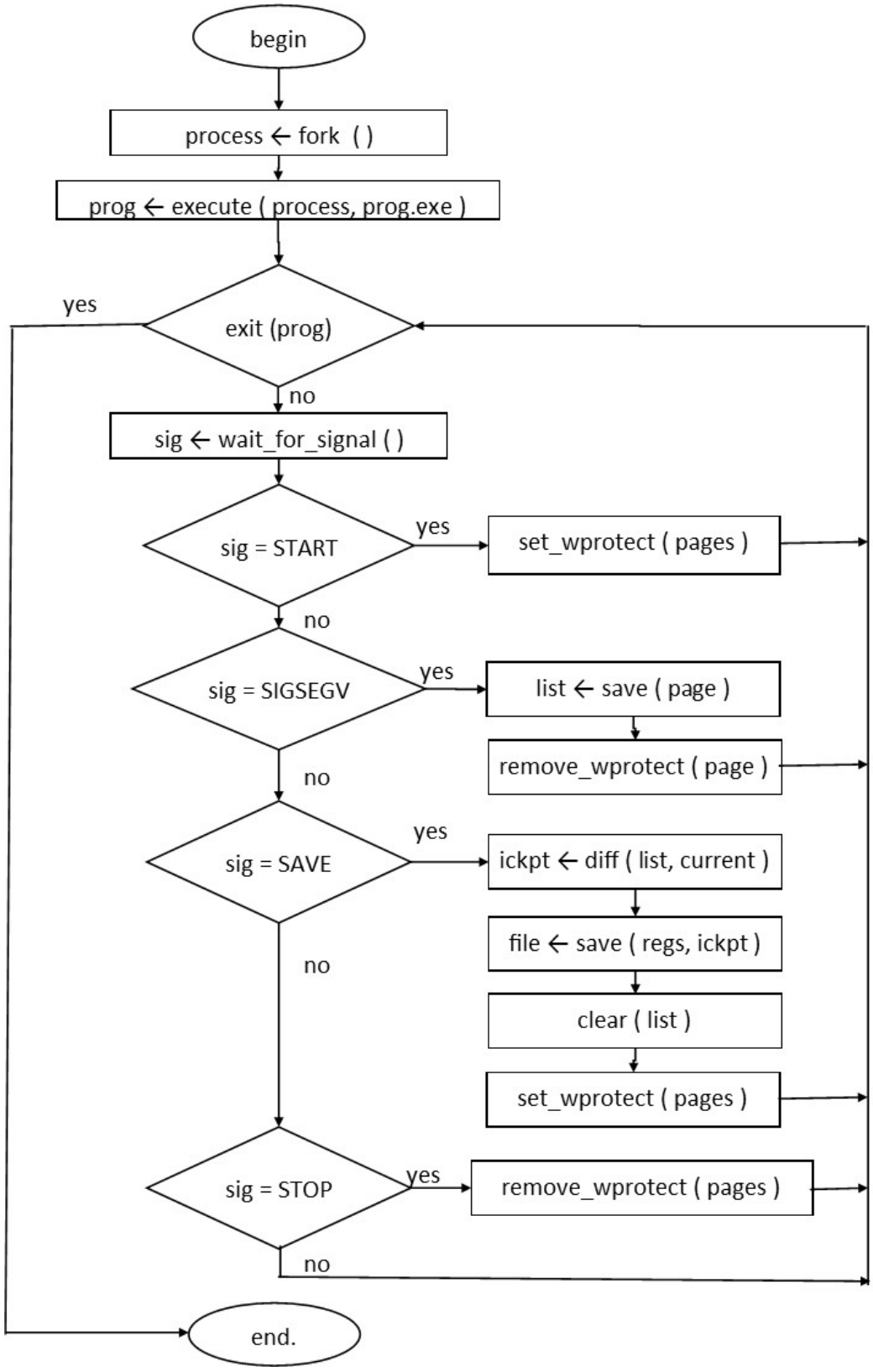

Figure 2.13: The principle of the DICKPT monitor.

Monitor: The Figure 2.13 presents the flowchart of the monitor based on DICKPT. At the beginning, the monitor calls and runs the program in a child process. It also tracks the program memory under the provision of the `ptrace` Linux system call. While the program is running, the monitor waits for the signals sent by the program. On the arrival of the signal, the monitor executes the relevant functions to generate DICKPTs. These functions are explained in the following:

- `set_wprtect(pages)`: sets all writable pages of the application memory to write protection. This function calls a system function provided by the driver.
- `remove_wprtect(pages)`: removes the write protection of an application memory page. It calls a system function provided by the driver.
- `save(page)`: reads application's memory by page address and saves it into a buffer. The read memory function is also implemented in the driver.
- `diff(list, current)`: compares the buffer with the current application's memory to find the modified data. The word-level granularity is implemented in DICKPT.
- `save(regs, ickpt)`: saves application's registers values and the modified data into checkpoint file. Register values are provided by a function implemented in the driver.
- `clear(list)`: clear the list of buffers.

Driver: At the kernel-level, a character device driver [85] has been implemented to provide system functions. This allows the monitor to manipulate the application memory more easily. The main functions of DICKPT's driver are:

- `lock_process_image()` sets all memory pages of the application process to write protection.
- `unlock_process_image()` sets all memory pages of the application process to their initial status.
- `unlock_a_page(addr)` sets a memory page of the application process indicated by `addr` to its initial status.
- `read_range(addr, length, dst)` reads `length` bytes of data from the process memory, starting at `addr`. The result is saved at the the `dst` address in the user space.
- `write_range(addr, length, values)` write `length` bytes data to the process memory, starting at `addr`. The written data are located at the `values` in the user space.

2.3.3.3 Data structure

The checkpoint generated by DICKPT contains two sets of data: a) register values – ie. all registers values at the checkpoint's time, and b) process's address space – ie. all modified data as compared to the previous checkpoint of the executing process.

For modified data, the memory granularity is world-level (4-byte word in this case), so in most of the cases a lot of space is needed to store the address of each word. In [14], authors have proposed four structures to optimize the checkpoint size:

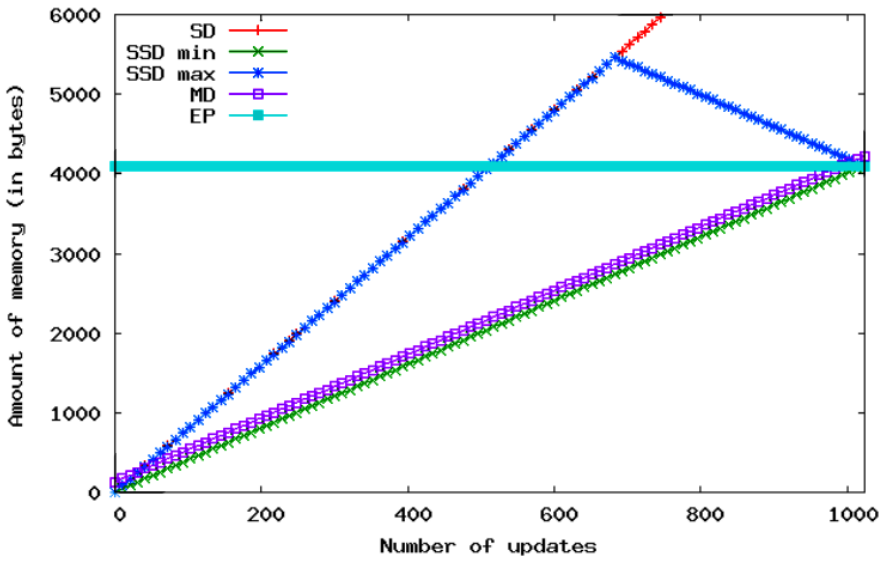

Figure 2.14: Amount of memory needed to store the modified data of a memory page.

- *Single data (SD).* For each modification of a word, the data are stored in the format of 8-bytes: 4 bytes for the address, and 4 bytes for the data. The structure of the data is `{(addr, value)}`.

- *Several successive data (SSD).* The data are stored in a structure of type `{(addr, size, values)}`. The checkpoint saves `values` of `size` bytes, from `addr` address into checkpoint file. In DICKPT, the maximum for `size` is the page size.

- *Many data (MD).* The data are stored in a structure of type `{(addr, map, values)}`. Here, `map` is presented using a single bit per memory location (per word). For 4-kB pages, the size of the `map` is 1024 bits.

- *Entire page (ED).* In this case, the whole page is stored, so the structure of the data is `{(addr, values)}`. The size is identified automatically by the page size.

Figure 2.14 presents the comparison of the amount of memory to store the modified data on a 4-kB page by the 4 data structures [14]. `SD`, `MD`, and `EP` only depend on the number of updates. `SSD` depends on the number of updates and the distribution of the updated memory locations. The best case ($\texttt{SSD}_{min}$) is achieved when the set of contiguous memory location has been updated is single and the worst case ($\texttt{SSD}_{max}$) occurs when the distributed memory location of updated memory is reached the maximum number of times.

Each structure has advantages and drawbacks depending on the number and location of the modifications on each memory page. For example, `EP` is the best solution if the modified region is always the whole page, `SD` is always the best choice if only few words are updated but not many, `MD` is really good if there are a lot of modifications and in different locations, while `SSD` is an interesting solution when the number of updates is smaller than 34. Currently, data structures of DICKPT implements the `SSD` structure.

2.4 Checkpointing Aide Parallel Execution (CAPE)

In the previous works, there were two versions of CAPE having been developed. The later one using DICKPT has shown the higher performance compared with the previous one using completed checkpoints [4]. This section, only focuses on the present CAPE ie. the later version.

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. Then, the generated CAPE source code is compiled using a traditional C/C++ compiler. At last, the binary code can be executed independently on any distributed-memory system supporting the CAPE framework. The different steps of the CAPE compilation process for C/C++ OpenMP programs are shown in Fig. 2.15.

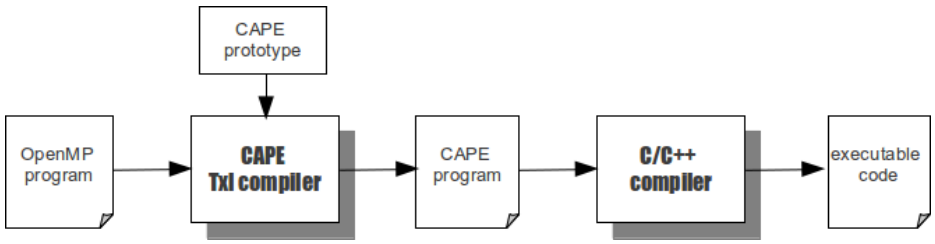

Figure 2.15: Translation of OpenMP programs with CAPE.

2.4.1 Execution model

The CAPE execution model is based on checkpoints that implement the OpenMP fork-join model. This mechanism is shown in Fig. 2.16. To execute a CAPE code on a distributed-memory architecture, the program first runs on a set of nodes, each node being run as a process. Whenever the program meets a parallel section, the master node distributes the jobs among the slave processes using the Discontinuous Incremental Checkpoints (DICKPT) [14, 4] mechanism. Through this approach, the master node generates DICKPTs and sends them to the slave nodes, each slave node receives a single checkpoint. After sending checkpoints, the master node waits for the results to be returned from the slaves. The next step is different since it depends on the nature of the node: the slave nodes receive their checkpoint, inject it into their memory, execute their part of the job, and send it back the result to the master node by using DICKPT; the master node waits for the results and after receiving them all, merges them before injection into its memory. At the end of the parallel region, the master sends the resulting checkpoint to every slaves to synchronize the memory space of the whole program.

2.4.2 System organization

In CAPE, each node consists of two processes. The first one runs the application program. The second one plays two different roles: the first as a DICKPT checkpointer and the second as a communicator between the nodes. As a checkpointer, it catches signals from the application process and executes appropriate handles. In the communicator role, it ensures the distribution of jobs and the exchange of data between nodes. Figure 2.17 shows the principle of this organization.

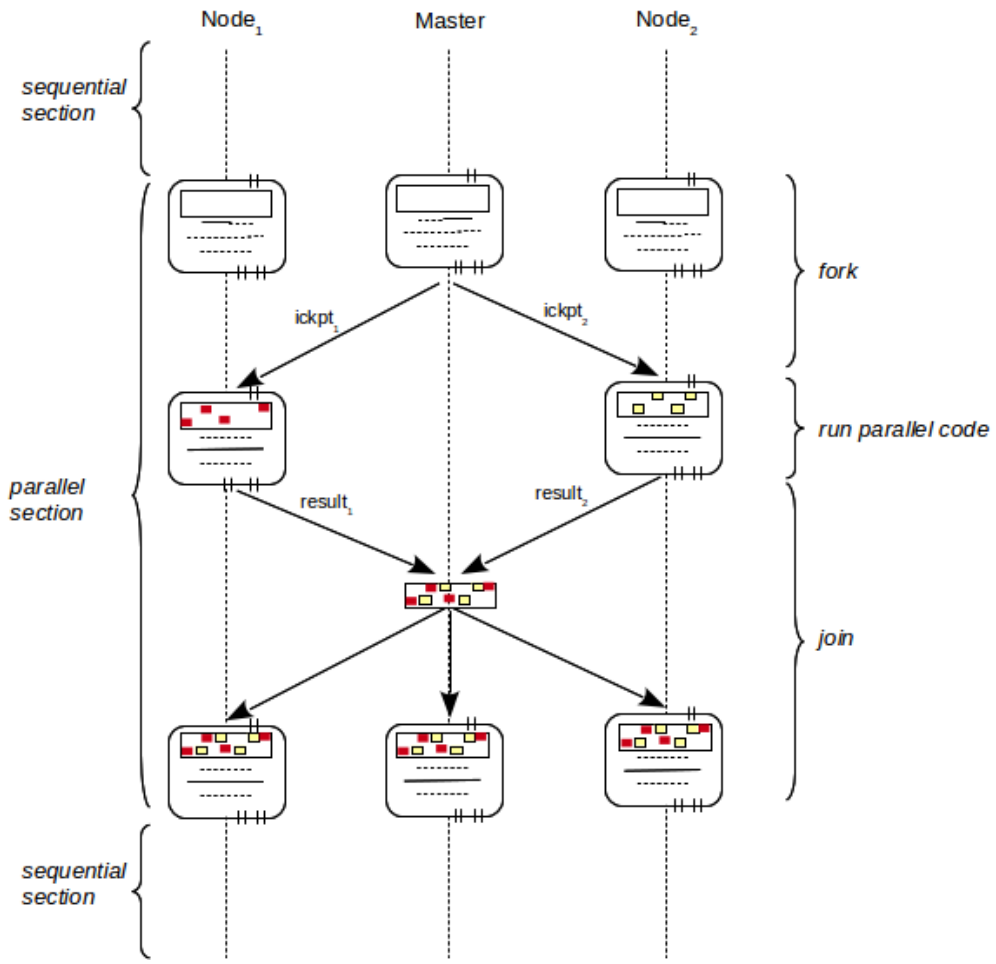

Figure 2.16: CAPE execution model.

In the current version, the master node is in charge of managing slave nodes and does not execute any application jobs in the parallel sections. Checkpoints are sent and received using sockets. The principle of sending and receiving is described as follows:

- The master node uses two sockets, the first one to listen to the requests from slave nodes, and the second one to send/receive checkpoints. The slave uses one socket to communicate with the master node.
- In order to send checkpoints from the master to the slave node, the monitor of the master maintains a loop to connect one by one with each slave. At each time, the master can only connect and send checkpoints to one slave.
- In order to receive checkpoints from the slaves, the master also sets up a connection with the slave, and receive the checkpoint from the current slave. The loop is created in order to connect and communicate with all slaves.

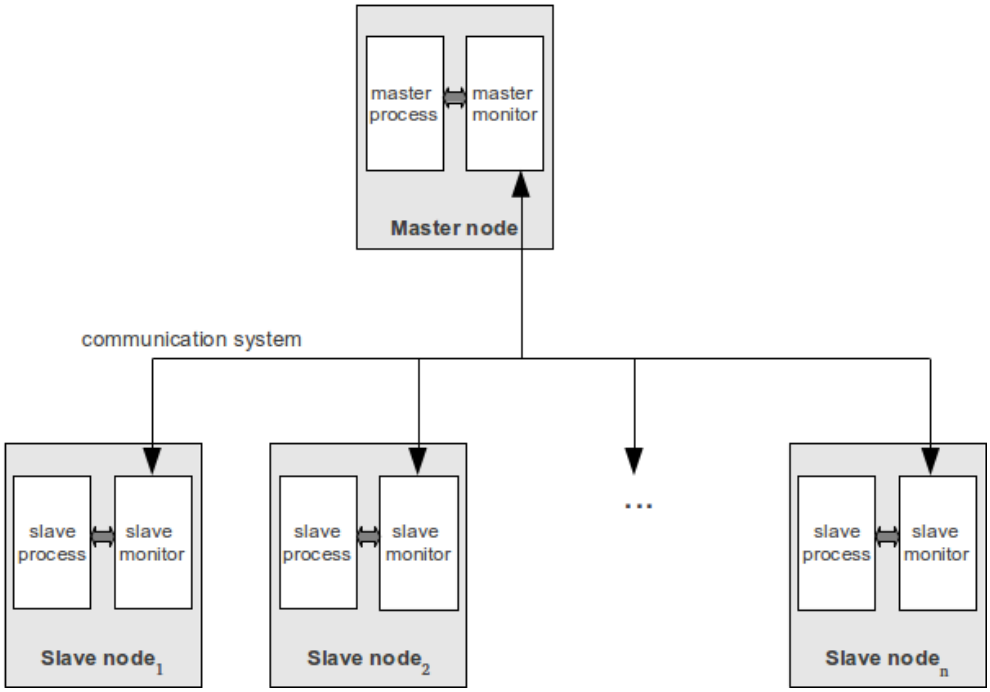

Figure 2.17: System organization.

2.4.3 Translation prototypes

To be compliant with OpenMP, CAPE provides a set of prototypes to translate OpenMP source code to CAPE source code (see Fig. 2.15). In work sharing constructs [2], only `pragma omp parallel for` and `pragma omp single` are implemented, the others are fist translated to `pragma omp parallel for`.

2.4.3.1 Prototype for the `pragma omp parallel for`

The prototype to translate `pragma omp parallel for` is presented in the Fig. 2.18. The fundamental CAPE's functions are described below:

- `start()`: associated with the `pragma dickpt start` directive as presented in Sec. 2.3.3. It sets write protection to all writable pages and starts or resumes checkpointing.
- `stop()`: associated with the `pragma dickpt stop` directive as presented in Sect. 2.3.3. It removes write protection to all write-protected pages and suspends checkpointing.
- `create(file)`: associated with the `pragma dickpt save` directive as presented in Sect. 2.3.3. It generates an incremental checkpoint and saves `file` to the buffer.
- `send(file)`: the master sends the incremental checkpoint indicated by `file` to all slave nodes.
- `broadcast(file, node)`: sends the content of `file` from the current node to `node`.
- `receive(file)`: the slave node receives the checkpoint sent by the master and save it into `file`.
- `wait_for(file)`: the master waits for receiving the incremental checkpoint files from all slave nodes and it merges these checkpoints to a `file`.
- `inject(file)`: injects the content of the incremental checkpoint indicated by `file` into the application process's memory. In CAPE, the program counter of each executing process is not updated by this value in the incremental checkpoint file.

2.4.3.2 Translation of the other directives to `pragma omp parallel for`

1) `parallel` directive. In OpenMP, this directive creates a team of threads and distributes the same region to all created threads in order to execute. However, on CAPE, the master does not execute any parts of jobs of the construct, and it is converted to `pragma omp parallel for` as presented in the Fig. 2.19.

2) `parallel sections` directive. In OpenMP, they contain a set of construct blocks. These blocks are distributed among and executed by the threads. In CAPE, it can be converted to `parallel for` before applying the existing prototype. Figure 2.20 presents an example of conversion of a `parallel sections` with three `section` blocks to a `parallel for`.

3) `for` and `sections` directives. These two work sharing constructs have to be declared inside the `parallel` construct. Therefore, `for` and `sections` are combined with a `parallel` construct to become `parallel for` and `parallel sections` constructs respectively.

```
# 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 2.18: Template for the *parallel for* with incremental checkpoints.

```
# pragma omp parallel
        D ;
```

↓ is converted to ↓

```
# pragma omp parallel for
for ( i = 0 ; i < number_of_slave_nodes - 1 ; i ++ )
        D ;
```

Figure 2.19: Form to convert `pragma omp parallel` to `pragma omp parallel for`.

```
# pragma omp parallel sections
{
        # pragma omp section
        P0 ;
        # pragma omp section
        P1 ;
        # pragma omp section
        P2 ;
}
```

↓ is converted to ↓

```
# pragma omp parallel for
for ( i = 0 ; i < 3 ; i ++ ) {
        switch ( i ) {
                case 0 :
                        P0 ;
                        break ;
                case 1 :
                        P1 ;
                        break ;
                case 2 :
                        P2 ;
        }
}
```

Figure 2.20: Form to convert `parallel sections` to `parallel for`.

Chapter 3

Optimization of chekpointing on CAPE

Contents

3.1 Motivation

Checkpointing techniques are the most important factors for the implementation of OpenMP on distributed-memory system using CAPE. CAPE-1 [3, 86] was implemented based on complete checkpoints, while CAPE-2 [4, 5] is based on incremental checkpoint. In spite of using different kinds of checkpoints, both CAPE-1 and CAPE-2 use checkpoints to divide jobs into work-sharing constructs and distribute instructions to slave nodes, and to automatically detect modified data after executing divided jobs on each slave node, then send them back to the master.

Discontinuous Incremental Checkpointing (DICKPT) [14] in CAPE-2 is a development based on incremental checkpoint containing two kinds of data, register's values and modified data of the process after executing a block of code. In which, the first one is copied from all register's values of the process at the checkpoint time, and the second one is identified based on write-protection techniques. The advantages of DICKPT over complete checkpoints on the two versions of CAPE, was shown through a comparison of the performance of CAPE-1 and CAPE-2 in [5].

In spite of the achieved results, the limitation of DICKPT mechanism on CAPE is the parallel regions in the OpenMP program that have to match the Bernstein's conditions [87]. This means the different nodes executing the divided jobs of a parallel region accessed to the different parts of data. Moreover, DICKPT detects all modified data in process's memory including shared and private variables, and saves them to checkpoint file. This does not only reduce the performance but also make wrong results for the program.

To solve these problems, we have proposed and developed an arithmetic model as well as a new mechanism of checkpointing. It was designed to implement the fork-join parallelism model, especially to apply to CAPE. The challenges which have to resolve in new checkpoint techniques are:

- Detecting the modified shared variables of the OpenMP program only.
- Storing data in checkpoint files with the data structure so that one can clearly identify the order the checkpoint files were generated on the different nodes.
- Proving the operations that can used to combine and find the different contents of the checkpoints.

In the results, we expect not only to reduce the checkpoint size and to improve the performance the implementation using checkpointing, but also overcome the limitations of OpenMP programs that need to match with Bernstein's conditions.

3.2 Arithmetic on checkpoints

Basing on the checkpointing techniques mentioned in Sec. 2.3, checkpoint contains the current information of the registers and a set of data for the process in memory. Each piece of data is informed by a word of the process's memory at a time. For complete checkpoint, it consists in a capture of all information in memory at the time of the generation of the checkpoint. For incremental checkpoints, it only contains the modified data in memory compared to the previous checkpoint. The checkpoints can be defined and denoted as follows.

3.2.1 Checkpoint definitions

Definition 1. Checkpoint

A Checkpoint contains 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, $t \in \{0, 1, 2, 3...N\}$.
- 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 $S_t = \{(a, v)_t\} = \{(a_1, v_1)_{t_1}, (a_2, v_2)_{t_2}, ..., (a_n, v_n)_{t_n}\}$.

Remark.

There cannot exist two memory elements having the same *address* in a Checkpoint.
If $S \in C$ and $(a, v)_t \in S$ then $(a, v')_{t'} \notin S$.

Definition 2. Empty memory element

There can exist a Checkpoint with no memory element. It is called the *empty* memory element. Denoted by ε.

Definition 3. Range of a Checkpoint

When a checkpoint is required to save the state of a process in specific ranges of memory, these ranges are called the range of the checkpoint. Denoted by:

$$R_C = \{list\_of\_memory\_of\_program\}$$

Definition 4. Equality of two Checkpoints.

Let $C' = (R_{t'}, S')$ and $C'' = (R_{t''}, S'')$ be two checkpoints. C' and C'' are equal if:

$$C' = C'' \equiv \begin{cases} R_{t'} = R_{t''} \\ S' = S'' \end{cases} \tag{3.1}$$

3.2.2 `Replacement` operation

3.2.2.1 Definition

Definition 5. A `replacement` operation is treated for 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:

$$(a, v_i)_{t_i} \odot (a, v_j)_{t_j} = (a, v_i \odot v_j)_{\max(t_i, t_j)} \tag{3.2}$$

where, $\odot$ can be any commutative and associative mathematical operations such as the OpenMP reduction operations are presented in Table 3.1 if it is provided by program, otherwise, it is calculated by:

$$v_i \odot v_j = \begin{cases} v_i & \text{if } t_i > t_j \\ v_j & \text{if } t_i < t_j \end{cases} \tag{3.3}$$

Operations ($\odot$)	**Initialization value**	**Description**
+	0	Addition
*	0	Multiplication
&	1	Binary AND operator
|	0	Binary OR operator
^	0	Binary XOR operator
&&	1	Logical AND operator
||	0	Logical OR operator

Table 3.1: OpenMP's reduction operations in C.

Remark. The `replacement` operation has following properties:

(i) *Communicative*:

$$v_1 \odot v_2 = v_2 \odot v_1 \tag{3.4}$$

Proof. From Equation (3.3), we have:

$$\begin{aligned} v_1 \odot v_2 &= \begin{cases} v_1 & \text{if } t_1 > t_2 \\ v_2 & \text{if } t_1 < t_2 \end{cases} \\ &= \begin{cases} v_2 & \text{if } t_2 > t_1 \\ v_1 & \text{if } t_2 < t_1 \end{cases} \\ &= v_2 \odot v_1 \end{aligned}$$

□

(ii) *Associative*:

$$(v_1 \odot v_2) \odot v_3 = v_1 \odot (v_2 \odot v_3) \tag{3.5}$$

Proof. From Equation (3.3), we have:

$$(v_1 \odot v_2) \odot v_3 = \begin{cases} v_1 & \text{if } (t_1 > t_2) \text{ and } \max(t_1, t_2) > t_3 \\ v_2 & \text{if } (t_1 < t_2) \text{ and } \max(t_1, t_2) > t_3 \\ v_3 & \text{if } \max(t_1, t_2) < t_3 \end{cases}$$

$$= \begin{cases} v_1 & \text{if } (t_1 > t_2) \text{ and } (t_1 > t_3) \\ v_2 & \text{if } (t_1 < t_2) \text{ and } (t_2 > t_3) \\ v_3 & \text{if } (t_1 < t_3) \text{ and } (t_2 < t_3) \end{cases}$$

$$= \begin{cases} v_1 & \text{if } t_1 > \max(t_2, t_3) \\ v_2 & \text{if } (t_2 > t_3) \text{ and } (t_1 < t_2) \\ v_3 & \text{if } (t_2 < t_3) \text{ and } (t_1 < t_3) \end{cases}$$

$$= \begin{cases} v_1 & \text{if } t_1 > \max(t_2, t_3) \\ v_2 & \text{if } (t_2 > t_3) \text{ and } (t_1 < \max(t_2, t_3)) \\ v_3 & \text{if } (t_2 < t_3) \text{ and } (t_1 < \max(t_2, t_3)) \end{cases}$$

$$= v_1 \odot (v_2 \odot v_3)$$

□

Definition 6. The `replacement` operation with *empty* memory element.

Let $(a, v)_t$ be a memory element and ε be the empty memory element, the `replacement` operation on memory element and empty memory element can be defined as follows:

$$(a, v)_t \odot \varepsilon = \varepsilon \odot (a, v)_t = (a, v)_t \tag{3.6}$$

3.2.2.2 Properties

Let $(a, v_1)_{t_1}$, $(a, v_2)_{t_2}$, and $(a, v_3)_{t_3}$ be memory elements of different checkpoints. We have:

(i) *Commutative* :

$$(a, v_1)_{t_1} \odot (a, v_2)_{t_2} = (a, v_2)_{t_2} \odot (a, v_1)_{t_1} \tag{3.7}$$

Proof. From Definition 5 we have:

$(a, v_1)_{t_1} \odot (a, v_2)_{t_2} = (a, v_1 \odot v_2)_{\max(t_1,t_2)}$

$\odot$ and `max` are *communicative* and *associative* mathematical operations. Thus,

$v_1 \odot v_2 = v_2 \odot v_1$

and

$\max(t_1, t_2) = \max(t_2, t_1)$.

Therefore,

$(a, v_1)_{t_1} \odot (a, v_2)_{t_2} = (a, v_1 \odot v_2)_{\max(t_1,t_2)} = (a, v_2 \odot v_1)_{\max(t_2,t_1)} = (a, v_2)_{t_2} \odot (a, v_1)_{t_1}$ □

(ii) *Associative* :

$$((a, v_1)_{t_1} \odot (a, v_2)_{t_2}) \odot (a, v_3)_{t_3} = (a, v_1)_{t_1} \odot ((a, v_2)_{t_2} \odot (a, v_3)_{t_3}) \tag{3.8}$$

Proof. From Definition 5, we have:

$((a, v_1)_{t_1} \odot (a, v_2)_{t_2}) \odot (a, v_3)_{t_3} = (a, (v_1 \odot v_2) \odot v_3)_{\max(\max(v_1,v_2),v_3)}$

$\odot$ and `max` are *communicative* and *associative* mathematical operations. Thus,

$(v_1 \odot v_2) \odot v_3 = v_1 \odot (v_2 \odot v_3)$

and

$\max(\max(v_1, v_2), v_3) = \max(v_1, \max(v_2, v_3))$

Therefore, $((a, v_1)_{t_1} \odot (a, v_2)_{t_2}) \odot (a, v_3)_{t_3}$

$= (a, (v_1 \odot v_2) \odot v_3)_{\max(\max(v_1,v_2),v_3)}$

$= (a, v_1 \odot (v_2 \odot v_3))_{\max(v_1,\max(v_2,v_3))}$

$= (a, v_1)_{t_1} \odot ((a, v_2)_{t_2} \odot (a, v_3)_{t_3})$

□

3.2.3 Operations on Checkpoints memory members

3.2.3.1 `Merging` a memory element into a Checkpoint memory member

Definition 7. `Merging` a memory element $e = (a_j, v_j)_{t_j}$ into a Checkpoint memory member $S = \{(a_1, v_1)_{t_1}, (a_2, v_2)_{t_2}, ..., (a_i, v_i)_{t_i}\}$ is denoted by $S \oplus e$ and can be defined as follows:

$$S \oplus e = \begin{cases} S \cup \{(a_j, v_j)_{t_j}\} \text{ if } \nexists (a_j, \bullet)_\bullet \in S \\ (S \setminus (a_j, v_i)_{t_i}) \cup ((a_j, v_i)_{t_i} \odot (a_j, v_j)_{t_j}) \text{ if } \exists (a_j, \bullet)_\bullet \in S \end{cases} \tag{3.9}$$

3.2.3.2 `Excluding` a memory element from Checkpoint memory member

Definition 8. `Excluding` a memory element $e = (a_j, v_j)_{t_j}$ from a Checkpoint memory member $S = \{(a_1, v_1)_{t_1}, (a_2, v_2)_{t_2}, ..., (a_i, v_i)_{t_i}\}$ is denoted by $S \ominus e$ and can be defined as follows:

$$S \ominus e = \begin{cases} S \text{ if } \nexists (a_j, \bullet)_\bullet \in S \\ S \setminus (a_j, v_i)_{t_i} \text{ if } \exists (a_j, \bullet)_\bullet \in S \end{cases} \tag{3.10}$$

3.2.3.3 `Merging` of Checkpoints memory members

Definition 9. 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:

$$\begin{aligned} S_1 \oplus S_2 = &\{(a_i, v_i)_{t_i} \in S_1 / \nexists (a_i, v_j)_{t_j} \in S_2\} \\ &\cup \{(a_j, v_j)_{t_j} \in S_2 / \nexists (a_j, v_i)_{t_i} \in S_1\} \\ &\cup \{(a_i, v_i)_{t_i} \odot (a_j, v_j)_{t_j} / \forall (a_i, v_i)_{t_i} \in S_1, \forall (a_j, v_j)_{t_j} \in S_2 \, and \, a_i = a_j\} \end{aligned} \tag{3.11}$$

Theorem 3.2.1. *The* `Merging` *operation of checkpoints memory members is communicative and associative.*

(i) *Communicative*: $S_1 \oplus S_2 = S_2 \oplus S_1$.

Proof. From Equation (3.11), we have:

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

`union`, `max`, and `replacement` operations are *communicative*, then:

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

□

(ii) *Associative*: $(S_1 \oplus S_2) \oplus S_3 = S_1 \oplus (S_2 \oplus S_3)$.

Proof. Let:

$A_1 = \{(a_i, v_i)_{t_i} \in S_1 / \nexists (a_i, v_j)_{t_j} \in S_2 \text{ and } \nexists (a_i, v_k)_{t_k} \in S_3\}$

$A_2 = \{(a_j, v_j)_{t_j} \in S_2 / \nexists (a_j, v_i)_{t_i} \in S_1 \text{ and } \nexists (a_j, v_k)_{t_k} \in S_3\}$

$A_3 = \{(a_k, v_k)_{t_k} \in S_3 / \nexists (a_k, v_i)_{t_i} \in S_2 \text{ and } \nexists (a_k, v_j)_{t_j} \in S_2\}$

$B_{12} = \{(a_i, v_i)_{t_i} \in S_1 / \exists (a_i, v_j)_{t_j} \in S_2 \text{ and } \nexists (a_i, v_k)_{t_k} \in S_3\}$

$B_{13} = \{(a_i, v_i)_{t_i} \in S_1 / \exists (a_i, v_j)_{t_j} \in S_3 \text{ and } \nexists (a_i, v_j)_{t_j} \in S_2\}$

$B_{21} = \{(a_j, v_j)_{t_j} \in S_2 / \exists (a_j, v_i)_{t_i} \in S_1 \text{ and } \nexists (a_j, v_k)_{t_k} \in S_3\}$

$B_{23} = \{(a_j, v_j)_{t_j} \in S_2 / \exists (a_j, v_k)_{t_k} \in S_3 \text{ and } \nexists (a_j, v_i)_{t_i} \in S_1\}$

$B_{31} = \{(a_k, v_k)_{t_k} \in S_3 / \exists (a_k, v_i)_{t_i} \in S_1 \text{ and } \nexists (a_k, v_j)_{t_j} \in S_2\}$

$B_{32} = \{(a_k, v_k)_{t_k} \in S_3 / ((\exists (a_k, v_j)_{t_j} \in S_2 \text{ and } \nexists (a_k, v_i)_{t_i} \in S_1\}$

$E_1 = \{(a_i, v_i)_{t_i} \in S_1 / \exists (a_i, v_j)_{t_j} \in S_2 \text{ and } \exists (a_i, v_k)_{t_k} \in S_3\}$

$E_2 = \{(a_j, v_j)_{t_j} \in S_2 / \exists (a_j, v_i)_{t_i} \in S_1 \text{ and } \exists (a_j, v_k)_{t_k} \in S_3\}$

$E_3 = \{(a_k, v_k)_{t_k} \in S_3 / \exists (a_k, v_i)_{t_i} \in S_2 \text{ and } \exists (a_k, v_j)_{t_j} \in S_2\}$

From Equation (3.11), we have:

$(S_1 \oplus S_2) \oplus S_3 =$
$(A_1 \cup A_2 \cup (B_{12} \odot B_{21})) \cup A_3 \cup (((E_1 \odot E_2) \odot E_3) \cup (B_{13} \odot B_{31}) \cup (B_{23} \odot B_{32}))$

As $\cup$ and $\odot$ operations are *communicative* and *associative*:

$(S_1 \oplus S_2) \oplus S_3 =$
$A_1 \cup (A_2 \cup A_3 \cup (B_{23} \odot B_{32})) \cup ((E_1 \odot (E_2 \odot E_3)) \cup (B_{12} \odot B_{21}) \cup (B_{13} \odot B_{31})) =$
$S_1 \oplus (S_2 \oplus S_3)$

□

3.2.3.4 `Excluding` of Checkpoints memory members

Definition 10. `Excluding` of Checkpoint memory member $S'' = \{(a_1, v_1)_{t_1}, (a_2, v_2)_{t_2}, \ldots, (a_j, v_j)_{t_j}\}$ from Checkpoint memory member $S' = \{(a_1, v_1)_{t_1}, (a_2, v_2)_{t_2}, \ldots, (a_i, v_i)_{t_i}\}$, is denoted by $S' \ominus S''$ and can be defined as follows:

$$S = S' \ominus S'' = S' \ominus \sum_{e \in S''} (e) \tag{3.12}$$

where, $e = (a_j, v_j)_{t_j}$ is a memory member of S''.

3.2.4 Operations on Checkpoints

3.2.4.1 `Merging` of Checkpoints

Definition 11. 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:

$$C = C_1 \oplus C_2 = (R_{\max(t_1, t_2)}, S_1 \oplus S_2) \tag{3.13}$$

Theorem 3.2.2. *The* `Merging` *operation on checkpoints is communicative and associative.*

(i) *Communicative*: $C_1 \oplus C_2 = C_2 \oplus C_1$.

Proof. From Definition 11, $C_1 \oplus C_2 = (R_{\max(t_1,t_2)}, S_1 \oplus S_2)$. It holds that: (i) $R_{\max(t_1,t_2)}$, and (ii) $S_1 \oplus S_2$.

For part (i), as `max` is a communicative operation, $R_{\max(t_1,t_2)} = R_{\max(t_2,t_1)}$.

For part (ii), as operator $\oplus$ is communicative (see Theorem 3.2.1), $S_1 \oplus S_2 = S_2 \oplus S_1$

Therefore, $C_1 \oplus C_2 = (R_{\max(t_2,t_1)}, S_2 \oplus S_1) = C_2 \oplus C_1$.

□

(ii) *Associative*: $(C_1 \oplus C_2) \oplus C_3 = C_1 \oplus (C_2 \oplus C_3)$.

Proof. From Definition 11, we have:

$(C_1 \oplus C_2) \oplus C_3 = \{(R_{\max(\max(t_1,t_2),t_3)}, (S_1 \oplus S_2) \oplus S_3)\}$.

It holds that: (i) $R_{\max(\max(t_1,t_2),t_3)}$, and (ii) $(S_1 \oplus S_2) \oplus S_3$.

For part (i), as `max` is associative operation, $R_{\max(\max(t_1,t_2),t_3)} = R_{\max(t_1,\max(t_2,t_3))}$.

For part (ii), as operator $\oplus$ is associative (see Theorem 3.2.1), $(S_1 \oplus S_2) \oplus S_3 = S_1 \oplus (S_2 \oplus S_3)$

Therefore, $(C_1 \oplus C_2) \oplus C_3 = (R_{\max(t_1,\max(t_2,t_3))}, S_1 \oplus (S_2 \oplus S_3)) = C_1 \oplus (C_2 \oplus C_3)$.

□

3.2.4.2 `Excluding` of Checkpoints

Definition 12. The `Excluding` operation of checkpoint $C_2 = (R_{t_2}, S_2)$ from checkpoint $C_1 = (R_{t_1}, S_1)$, denoted by $C = C_1 \ominus C_2$, is a checkpoint that can be presented as follows:

$$C = C_1 \ominus C_2 = (R_{\max(t_1,t_2)}, S_1 \ominus S_2) \tag{3.14}$$

3.2.5 Conclusion

In this section, we present data structure of a checkpoint and mathematical model to represent and computer on them. The timestamp is very useful in determining the order of checkpoints, especially when they are generated by a program that is executed in parallel on different processes. The operations provide a mechanism for calculating directly on checkpoints regardless of the order of the checkpoints being generated, the overlapping of the address space on checkpoints, or the operations used in the program to merge results after computing in parallel like `reduction()` clause of OpenMP. That reduces the checkpoint size and the time it takes to gather checkpoints. The next part of this chapter presents the application of the arithmetic on checkpoints in CAPE as a case study.

3.3 Time-stamp Incremental Checkpointing (TICKPT)

To implement the arithmetic on checkpoints proposed above, we have developed a new checkpoint technique called Time-stamp Incremental Checkpointing (TICKPT). TICKPT is an improvement of DICKPT that adds a new factor – timestamp – into incremental checkpoints and removes unnecessary data based on selecting only the modified data of shared variables of the OpenMP program.

Basically, TICKPT contains three mandatory elements including register values, modified regions of shared variables in the memory of the process, and their timestamp. As well as DICKPT, in TICKPT, the register values are extracted from all registers of the process in the system. A time-stamp is a value that identifies the order of checkpoints in a program generated for different processes when the program is executing in parallel. The challenge with this method is the identification of the timestamp, and shared variables, and the detection of modified region of shared variables in the process memory.

3.3.1 Identifying the time-stamp

Time-stamp is one of the most important elements of TICKPT. It is used to identify the order of updates at the same memory address in different virtual address spaces by the different processors, and the last register values of the program when it is executing.

The concept of activation tree [88] can be used to represent the execution order of procedures during the execution of a program. Each node is associated with one activation tree. The root is the activation of the `main` function. At a node, for an activation of procedure `p`, the children correspond to the activation of the procedures called by this activation of `p`. As a result, the order of these activations are obtained by reading the tree from left to right. Therefore, it can be used to identify the sequence of function called in a program.

However, in particular cases when applied on CAPE, the program is initialized on all processors at the begin, and checkpoints are always generated at the synchronization points that are declared at the same level in function calls. It has been implemented using a simple method that depends on the situations following:

- If each processor executes a different part of the program, the timestamp is identified by the value of the program counter before calling checkpoint generation.
- If each processor executes a different part of a loop of the program, the timestamp is the last iterator of each part dedicated to the processor.
- If each processor executes exactly the same piece of code in the program, the timestamp is value `processor_id` complemented to `total_processors - 1`.

3.3.2 Variable analysis

In TICKPT, the program variables are monitored. Variables declared in the global scope of the program and in activation functions of the activation tree are marked as live variables. Otherwise, they are not live variables. Live variables are managed by the monitor.

In an OpenMP program, data-sharing variable attributes can be set up either implicitly or explicitly [2]. Figure 3.1 presents the location of these variables in the virtual address space of a process. The default context of shared variables is determined by reaching one of the following conditions:

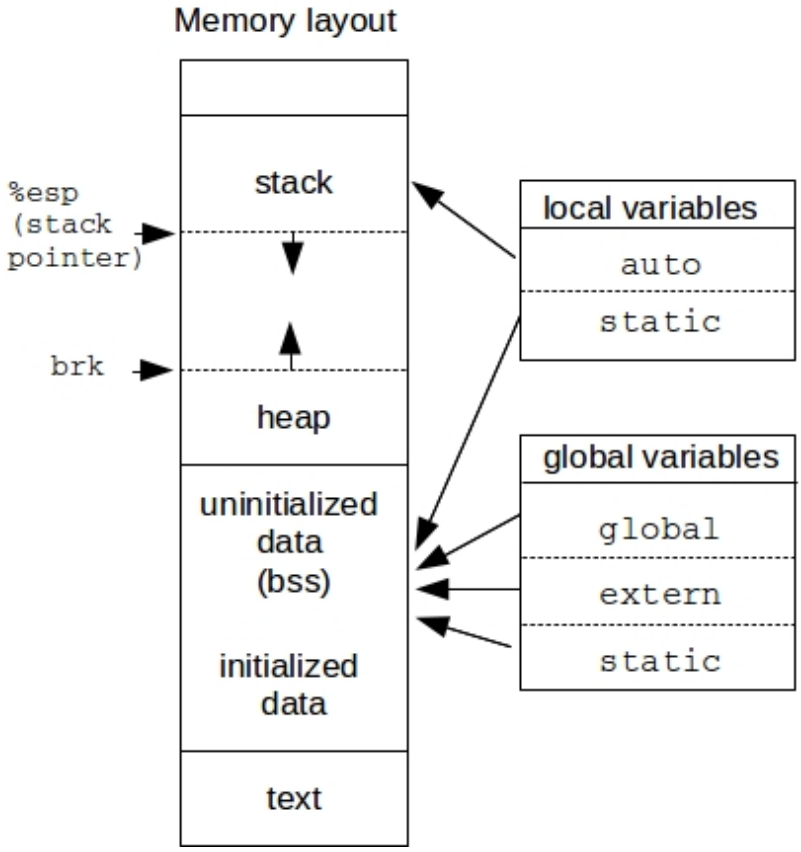

Figure 3.1: Allocation of program variables in virtual process memory.

- Global variables are stored in the `data` segment.
- Variables are declared using the `static` keyword allocated in the `data` segment.
- Dynamic variables are allocated in the `heap` segment.
- The other variables (automatic and local variables) are declared outside any parallel constructs are allocated in the `stack`.

These variables can be detected at the running time by a function provided by the checkpoint monitor. These functions save and manage the *address*, *length*, and *attributes* of implicitly shared variables.

To explicitly change the status of a variable, the programmer can use data-sharing attributes like OpenMP directive `#pragma omp thread private (list of variables)` or relative clauses. Therefore, it is easy to detect and modify the attributes of variables that are managed by the monitor. The OpenMP data-sharing clauses are reminded in Table 3.2.

Clauses	**Description**
default(none|shared)	Specifies the default behavior of variables
shared(list)	Specifies the list of shared variables
private(list)	Specifies the list of private variables
firstprivate(list)	Allows to access the value of the list of private variables when entering parallel region
lastprivate(list)	Allows to share the value of the list of private variables when exiting the parallel region
copyin(list)	Allows to access the value of threadprivate variables
copyprivate(list)	Specifies the list of private variables that should be shared among all threads.
reduction(list, ops)	Specifies the list of variables that are subject to a reduction operation at the end of the parallel region.

Table 3.2: OpenMP data-sharing clauses.

3.3.3 Detecting the modified data of shared variables

For the memory members of checkpoints, as well as for the incremental checkpointing presented in Sec. 2.3.2 and 2.3.3, TICKPT has to identify the modified regions in the process's memory. We have implemented and tested two methods: page write-protection, and duplication of shared variables. For both methods, it is only modified data of shared variables that is detected.

3.3.3.1 Method 1: page write-protection mechanism

Similar to DICKPT as presented in Sec 2.3.3, TICKPT - Method 1 uses the page write-protection mechanism to detect the modified pages when executing the program. However, instead of setting write-protection to all pages in the virtual memory of the process as DICKPT does, TICKPT does it to all pages containing shared variables only. Data of these pages are also copied when `SIGSEGV` signals occur. During the executing of the program, the attribute of shared variables can be changed by runtime functions, and these status are saved. When reaching `pragma tickpt save`, the copied data is compared with the current data at the same location in the virtual memory, and the modified data of shared variables are selected to write to the checkpoint file.

To save this part, we have use a SSD structure (see Sec 2.3.3.3). The modified data in each page is saved in the form of `{(address, len, data)}` triples, in which a maximum value of *len* bytes is the size of the page.

3.3.3.2 Method 2: shared-variables duplication

In this method, to avoid the sad effect on the performance of the program using TICKPT based on page write-protection mechanism, a shared-variables duplication method was implemented. When reaching `pragma tickpt start`, instead of setting write-protection to all pages that contain shared variables like TICKPT - Method 1 does, the value of these variables are copied. As well as TICKPT - Method 1, the attribute of shared variables can be updated by using the runtume library. At the `pragma tickpt save`, the copied data are compared with the current data, and the modified parts of shared variables are written to the checkpoint file. The buffer is cleared whenever the program meets `pragma tickpt stop`.

The SSD structure (see Sec 2.3.3.3) is used to store this part of checkpoint. However, unlike the previous method, the maximum value for `len` is not limited to the page size, but the maximum size of variables in the program.

3.4 Analysis and Evaluation

This section analyses the contribution of Arithmetics on Checkpoint for the implementation of fork-join model to overcome the limitations of the previous implementation. The result of experimentations are presented in next chapter. Besides, advantages and drawbacks of both methods implemented in TICKPT, as well as a comparison with DICKPT are analyzed and evaluated below.

3.4.1 Contributions of Arithmetics on Checkpoints

Arithmetics on Checkpoints as presented in Section 3.2 have a very important role in the optimization of checkpointing, especially when applied to implement the fork-join model. It provides a theoretical model to implement and improve checkpointing on CAPE that overcomes the limitations of Bernstein's condtions. Moreover, based on this model, the checkpoint size and the communication time are significant reduced.

Considering an implementation of fork-join model is based on checkpointing on distributed memory systems that shows in Figure 3.2. In the previous implementations, the complete checkpoint in CAPE-1 and DICKPT in CAPE-2 (see Sec. 2.4.1, p. 33), the OpenMP program is required to match with Bernstein's conditions. This means that each node in the

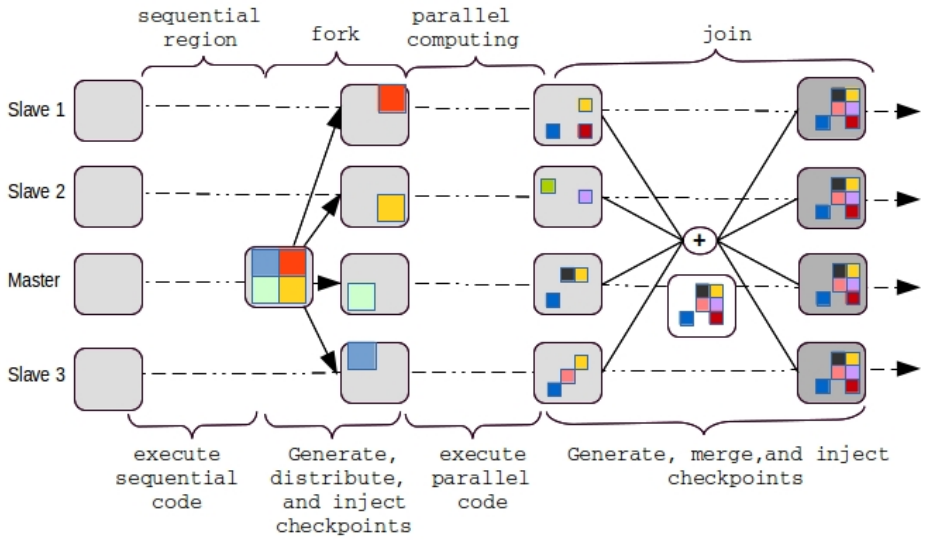

Figure 3.2: An implementation of the fork-join model based on checkpointing.

system has to modify different locations of the whole program address space to ensure the result of the execution is correct.

In this new theoretical model, we proposed to add a *timestamp* – a new element into checkpoints (see Definition 1, p. 41). A timestamp is an element in checkpoint that can be used to identify the order of data were modified, and develop the checkpoint operations. Thank to this model, which allows data at a single location to be modified at different nodes. Checkpoint operations are responsible for merging them at synchronization points and ensure the correctness of the result. Moreover, this model allows the use of these operations to implement the OpenMP `reduction` clause to be computed directly in checkpoint memory members.

For instance, considering the execution of an OpenMP code as shown in Fig. 3.3 on a 4-node cluster is based on the proposed model of checkpointing. This parallel code does not match the Bernstein's conditions. There is only 1 shared variable (`sum`). After executing the divided jobs, each node generates a checkpoint that contains the value of `sum` only in its memory member by the form of `(address, len, value)`. The OpenMP `reduction(+:sum)`computes the sum of `sum` values from 4 nodes. In this case, the `merging` checkpoint operation provided in Definition 11 implicitly implements the operation.

Regarding communication aspects, the previous model use the master node to store the lastest status of registers. The master node does not take any parts in divided jobs. The result checkpoints from the slave nodes are sent back to the master. After merging, this checkpoint is distributed to all slave nodes to resume the execution. Let p be the number of nodes in the system, n (in byte) be the size of checkpoints in each slave node, t_1 and t_2 be the latency times for sending/receiving 1 byte of data over the network, respectively. In this modeling we ignoring the merging time and assume that the number of nodes is power

```
int cal_pi(int n){
        double sum = 0.0, x = 0.0 , aux, step;
        int i;
        step = 1.0 / (double) n;
        #pragma omp parallel private(i,x,aux) shared(sum)
        {
                #pragma omp for reduction(+ : sum)
                for(i=0; i< n; i++){
                        x = (i+ 0.5) * step;
                        aux = 4.0/(1.0 + x*x);
                        sum += aux;
                }
        }
        return (step * sum);
}
```

Figure 3.3: An example of OpenMP program using a `reduction()` clause.

of two. The total time of synchronization in this case is:

$$\begin{aligned} T_{comm_1} &= 2(p-1)t_1 + (p-1)nt_2 + (p-1)n(p-1)t_2 \\ &= 2(p-1)t_1 + p(p-1)nt_2 \\ &= (p-1)(2t_1 + npt_2) \end{aligned} \tag{3.15}$$

In the new theoretical checkpointing model, checkpoints can be merged together without requirement of ordering. The order checkpoints are considered is saved in each of them, so this can be performed by using the Recursive Doubling algorithm [89] as illustrated in Fig. 3.4.

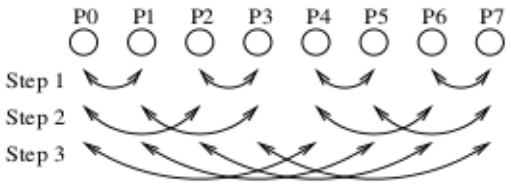

Figure 3.4: The Recursive Doubling algorithm.

At the first step the amount of transferred data is n bytes. At the second step, it is $2n$ bytes, etc., and after step k, it is $(2^k - 1)n$ bytes. The total number of steps in this case is $2^{\log(p)}$. Therefore, the time of synchronization is:

$$\begin{aligned} T_{comm_2} &= \log(p)t_1 + (1 + 2 + \cdots + 2^{\log(p)-1})nt_2 \\ &= \log(p)t_1 + (p-1)nt_2 \end{aligned} \tag{3.16}$$

From Equation (3.15) and (3.16), the new theoretical model provides a mechanism that significantly reduce the communication time at the join phase.

3.4.2 Detection modified data with TICKPT

In terms of checkpoint size, both methods use the SSD structure based on triplet `(address, len, data)` to store the modified value of shared variables. However, for a continuously modified memory area that is larger than the page size, the first method divides it to page size, but not the second. Therefore, with more than one continuous page of modified data, the first method takes at least 8 bytes than the second one to store the same data.

Another aspects is that setting write-protection to virtual memory and handling `SIGSEGV` signal with the first method reduces the performance of the program. In order to compare the size of checkpoint memory members of two methods and the effect on the performance of the program, some experiments have been done.

```
#define N 100000
int A[N][N], B[N][N], C[N][N];
int sum_vector(int n){
        int i, j;
        #pragma omp parallel private(A,B) shared(C)
        {
                #pragma omp for nowait
                for(i=0; i< n; i++)
                        for(j=0; j< n; j++){
                                A[i][j] = rand() % 100;
                                B[i][j] = rand() % 100;
                        }
                #pragma omp for nowait
                for(i=0; i< n; i++)
                        for(j=0; j< n; j++)
                                C[i][j] = A[i][j] + B[i][j];
        }
return 0;
}
```

Figure 3.5: OpenMP program to compute the sum of two matrices.

As can be seen in Figure 3.5, the program was used to do the experiments that computes the sum of two two matrices composed of random integer numbers. The block of code in `omp parallel` is taking checkpoints. The size of matrices are varied to compare the size of checkpoint memory members for the two methods, and the effect on performance of the program. These experiments were performed on a Notebook includes 4x Intel(R) Cores(TM) i5 (3rd Gen) 3320M CPU@2.6GHz and 4GB of RAM. The machine is operated with the Ubuntu 14.04.5 LTS 32-bit system.

Table 3.3 shows the size of checkpoint memory members (in bytes) that are generated by the two methods. Both methods recognize shared memory area containing matrix C, and private areas that containing the variables i, j, A, and B. Therefore, only memory area of C is handled to detect the modified data.

As analyzed above, if the continuous modified data in memory are larger than the page size, method 1 breaks them into page size segment to store them into the checkpoint. Therefore, it takes more space than method 2 to store relevant data. Hence, in the case of

N	TICKPT - Method 1	TICKPT - Method 2	Difference	Percentage (%)
10	408	408	0	0.0
100	40088	40008	80	0.20
1000	4007824	4000008	7816	0.20
2000	16031256	16000008	31248	0.19
3000	36070320	36000008	70312	0.19
4000	64125008	64000008	125000	0.19
6000	144281256	144000008	281248	0.19
8000	256500008	256000008	500000	0.19

Table 3.3: Size of checkpoint memory members (in bytes) for method 1 and 2.

continuous modified data less than the page size (eg. $N = 10$), the size of checkpoint memory member is equal for the two methods. In the other cases, method 2 is around 0.19% smaller than with method 1.

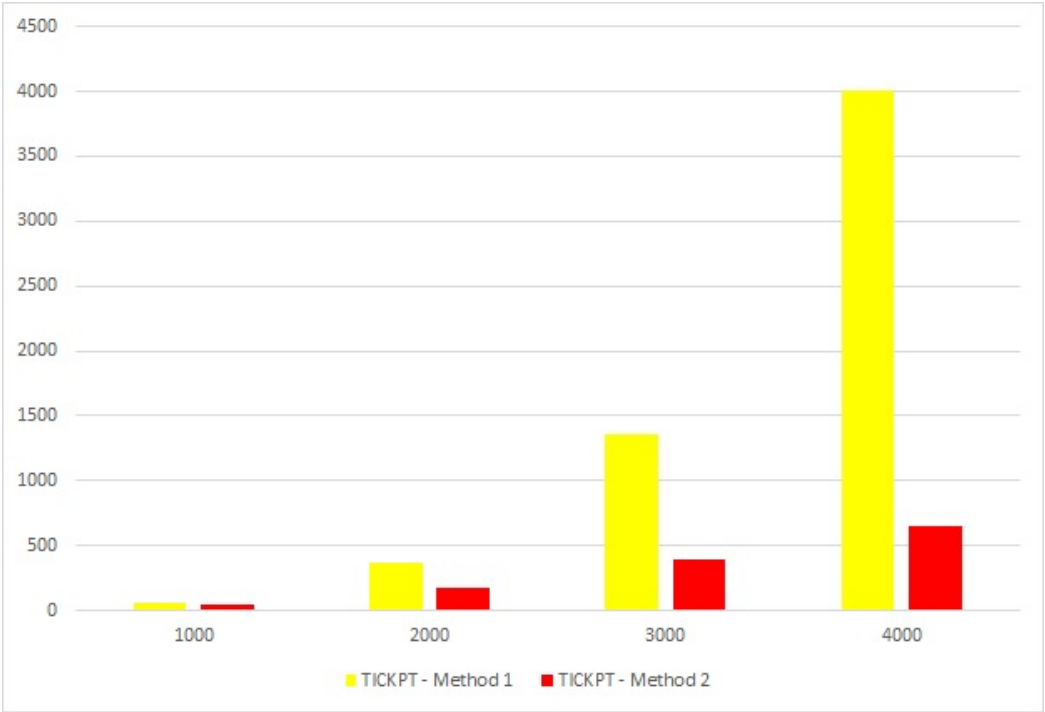

Figure 3.6: Execution time (in milliseconds) for method 1 and 2

The execution times of programs using both methods of TICKPT are presented in Fig. 3.6. When method 1 is used, a series of operations such as set write-protection, catching `SIGSEGV` signals, coping data, removing write-protection, re-writing data to the page, etc. takes a lot of time. As a result, in the Fig. 3.6, the larger the number of modified pages, the longer time. For method 2, shared variables are copied at the `start` pragma of checkpointing and no time is spent in operations like method 1. Experimental results show that the

execution time using method 1 is much higher as compared with method 2.

The drawback of method 2 is that it takes more space to store shared variables data at the begin blocks to checkpoint. However, this drawback can be handled by programmers in OpenMP programs by explicitly setting variables attributes.

From the analysis and experiments above, we decided to use the duplication of shared variables for CAPE. From now on, when referring to TICKPT, only method 2 is considered.

3.4.3 TICKPT vs. DICKPT

TICKPT is an improvement of DICKPT that consist in adding time-stamps in checkpoints, selects only modified values of shared variables, and uses the duplication of shared variable method to detect modified values. This checkpointing method supports the implementation of arithmetic on checkpoint as presented in Sec. 3.2.

To compare checkpoint sizes and the effect on the execution time of the program, some experiments were conducted using the same program and environment as presented in Sec. 3.4.2. As the size of register members is equal for both TICKPT and DICKPT, only checkpoint memory members generated by both techniques have been measured.

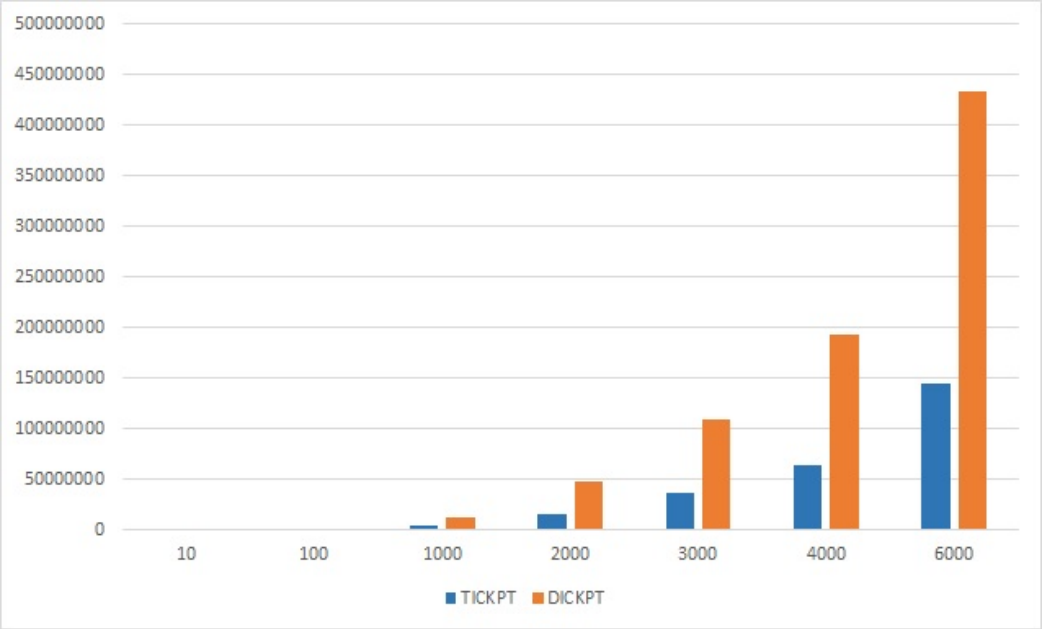

Figure 3.7: Size of checkpoint memory member (in bytes) for TICKPT and DICKPT.

Fig. 3.7 presents the size of checkpoint memory members generated by two techniques. The size of checkpoint memory member of DICKPT are 3 times larger than the one with TICKPT. This is due to the fact that DICKPT detects the modified values of the whole memory allocated to the program. It includes variables i, j, A, B, and C, whereas TICKPT only identifies the modified data of shareable variables – typically variable C in this case.

To assess the impact of the different checkpointing techniques to the program's execution time, we measured and compared the execution time for three situations: program not using checkpoints, program using TICKPT, and program using DICKPT. Table 3.4 presents the results of these measurements for different sizes of N. It shows that the impact of TICKPT

N	Without Checkpointing	Using TICKPT	Using DICKPT
1000	36	45	142
2000	136	168	1528
3000	307	391	8210
4000	536	649	38790

Table 3.4: Execution time (in milliseconds).

to execution time of program on this situation is around 20-25%, while with DICKPT it is more than 290%. In fact DICKPT sets write-protection to all memory pages allocated to execute the program, handles *SIGSEGV* signal, removes write-protection, etc. These operations affect the execution time of the program for too much.

Indeed, the analysis and experiments show the outstanding advantages of TICKPT over DICKPT. Hence, using TICKPT instead of DICKPT in CAPE improves its performance and capability. Furthermore, TICKPT supports arithmetic on checkpoints, that improves the reliability of the program.

3.5 Conclusion

To sum up, this chapter presented arithmetics on checkpoints, including the definitions and operations that can performed on checkpoints. This theoretical model allows checkpoints to be merged, find the difference without any requirements like matching with Bernstein's conditions, etc. This contributes to improve the capability and reliability of the system to port OpenMP on distributed memory architectures based on checkpointing.

In addition, the analysis showed that the theoretical model is able to provide a mechanism for merging checkpoints in different nodes without requirement of checkpoint's ordering. We also developed TICKPT, an improvement of DICKPT, which can uses checkpoint operations. TICKPT was implemented using two methods to compare and select the best one for CAPE. The analysis and experiments results show that checkpoints generated by TICKPT are significantly reduced as compare with DICKPT. Moreover, the performance of the program using TICKPT is not impacted as much. As a result, TICKPT improves the performance of CAPE.

Chapter 4

Design and implementation of a new model for CAPE

Contents

4.1 Motivation

CAPE have been using DICKPT 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 provide high performance. However, there are some limitations:

- OpenMP programs that work on CAPE must fulfill with Bernstein's conditions. This is reason why the matrix-matrix product has been extensively use 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 (see Fig. 2.16).
- 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 mater. The master receives, merges checkpoints together and sends the result back to the slave nodes in order to synchronize data (see Fig. 2.16).

This chapter presents the design and implementation of a new model for CAPE based on TICKPT to bypass the drawbacks shown above. The new implementation based on TICKPT improves the performance, capability, and reliability of this solution.

4.2 New abstract model for CAPE

Figure 4.1 presents the new abstract model for CAPE. It is designed based on TICKPT and uses MPI to transfer data over the network.

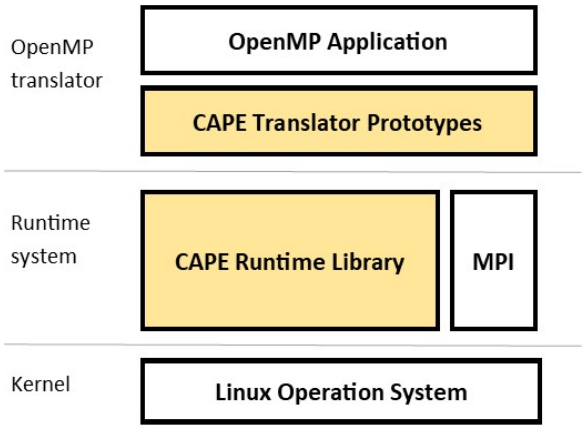

Figure 4.1: New abstract model for CAPE.

As presented in the previous version, 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. The steps for the translation and the compilation are presented in Fig. 2.15.

In new the 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. The details of these prototypes are presented in Sec. 4.5.

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 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.

In new version, the page-fault mechanism is no longer used. Therefore, the driver included in the kernel no longer exists. This definitively improves the portability of CAPE.

4.3 Implementation of the CAPE memory model based on the RC model

As presented in Sec. 2.1.1.2, OpenMP uses the Relaxed-Consistence (RC) memory model. This model allows shared memory allocated in local memory of 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.

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 assessed 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. 4.2. Here, the `all_reduce()` function is responsible for gathering and merging the checkpoints generated by `generate_checkpoint()` function. The gathering and the merging is implemented using both Ring and Recursive Doubling algorithm illustrated in Fig. 4.3 and Fig. 4.4 respectively. Ring or Recursive Doubling algorithm is automatically selected to execute by the system depending on the size of the checkpoint.

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

Figure 4.2: Operations executed on `cape_flush()` function calls.

```
ring_allreduce(int id, int nnodes){
        left ← (id - 1 + nnodes) % nnodes;
        right ← (id + 1) % nnodes;
        C ← C_id;
        M ← C_id ;
        for(i = 1; i < nnodes; i++){
                Send M to right ;
                Receive M from left ;
                C ← merge(C, M, [operators]);
        }
        return C ;
}
```

Figure 4.3: Ring algorithm to gather and merge checkpoints.

```
recursive_doubling_allreduce(int id, int nnodes){
        nsteps ← log(nnodes);
        C ← C_id;
        for(i = 0; i < nsteps; i++){
                partner ← id XOR (1 << i);
                M ← C ;
                Send M to partner ;
                Receive M from partner ;
                C ← merge(C, M, [operators]);
        }
        return C ;
}
```

Figure 4.4: Recursive Doubling algorithm to gather and merge checkpoints.

- `cape_barrier()` sets a barrier and update shared data between nodes. This function calls `MPI_Barrier()` of the MPI runtime library, and then uses `cape_flush()` to update shared data.

4.4 New execution model based on TICKPT

Figure 4.5 illustrates the new 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 belong to any other constructs is executed in the same behavior at all nodes.
- When the program reaches a `parallel` region, at 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 all data 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 constructs, 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 in `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 [2], a `flush` is implicit at the following locations:
  - At the `barrier`
  - At the entry to and the exit from `parallel`, `critical`, and `automic` 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 calculates 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 Recursive Doubling

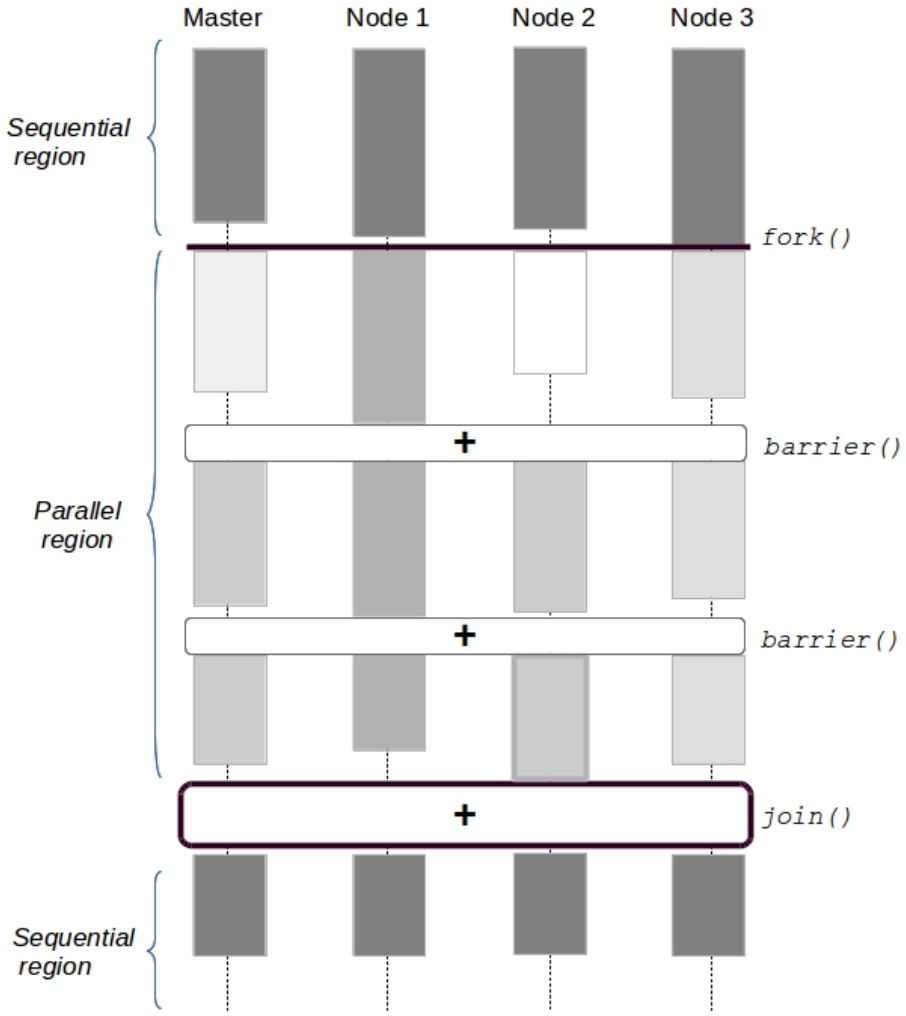

Figure 4.5: The new execution model of CAPE.

algorithm [89] as illustrated in Fig. 3.4. This allows CAPE to void the bottleneck and improve the performance of communication tasks as the analysis presented in Sec. 3.4.1.

With the features of TICKPT, checkpoints are possible to use checkpoint's operations presented in Sec. 3.2. This allows memory elements to have the same address when computing during merging. Therefore, it allows CAPE to work without the need for the program to match with 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 efficiency of the system.

The main drawback of this implementation is that `dynamic` and `guided` scheduling of work-sharing construct have not been implemented yet. However, they can be translated into `static` scheduling.

4.5 Transformation prototypes

To execute on distributed memory system under 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 on CAPE. This translation works under the provision of a set of CAPE prototypes.

At present, CAPE prototypes are totally different from the previous ones in order to adapt the new execution model based on TICKPT. Moreover, it allows CAPE to translate into the work-sharing constructs and other ones which are placed inside the `omp parallel` construct. However, it does not support nested `omp parallel` construct yet.

	BT	CG	EP	FT	IS	LU	MG	SP
parallel	2	2	1	2	2	3	5	2
for	54	21	1	6	1	29	11	70
parallel for		3					1	
master	2	2	1	10	4	2	1	2
single		12		5		2	10	
critical			1	1	1	1	1	
barrier				1	2	3	1	3
flush						6		
threadprivate			1					

Table 4.1: Directives used in the OpenMP NAS Parallel Benchmark programs.

The current work focuses on the implementation of OpenMP runtime functions, popular directives and their clauses. There have been considered as they are the most used in NAS parallel benchmark programs [90, 91]. The number of occurrences of these constructs is shown in Table 4.1.

Based on the general syntax of OpenMP directives as presented in Fig. 2.3, the general form of CAPE prototypes was designed and is illustrated in Figure 4.6. They are explained in the following:

- `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 functions. Depending on this label, the `cape_barrier()` function is called to update the shared data of the system. `parameter-1` and `parameter-2` are used to store the range of iterations for `for` loops, otherwise they both are assigned to zero. The `reduction-flag` is assigned to `TRUE` if there was a declaration of OpenMP reduction clause, otherwise it is `FALSE`.

- The `cape_clause_functions` is a set of CAPE runtime functions which is used to implement OpenMP clauses. These functions are presented in detail in Chap. 5.

- `ckpt_start()` marks the location where to start the checkpointing. When reaching the `ckpt_start()` function, the value of shared variables is copied.

```
cape_begin(directive-name, parameter-1, parameter-2);
        [cape_clause_functions]
        ckpt_start();

        //code blocks

cape_end(directive-name, reduction-flag );
```

Figure 4.6: General form of CAPE prototypes in C/C++.

Some directives are transformed by irregular form. The details of there prototypes and their functions is presented below.

4.5.1 The `parallel` construct

`omp parallel` is the fundamental construct of OpenMP. It aims at creating a set of threads to execute a code region in parallel. In CAPE, the program is already executed in a set of nodes, hence the `cape_begin()` function is called to record the list of shareable variables and their properties. After executing the jobs in the parallel region, the `cape_end()` function calls the `cape_barrier()` function and performs a set of operations to synchronize the shared variables among nodes and clean unnecessary data of the execution memory.

```
# pragma omp parallel [clauses]
        D ;
```

↓ is translated to ↓

```
cape_begin(PARALLEL, 0, 0);
        [cape_clause_functions]
        ckpt_start();
                D;
cape_end(PARALLEL, reduction-flag);
```

Figure 4.7: Prototype to transform the `pragma omp parallel` construct.

4.5.2 Work-sharing constructs

An OpenMP work-sharing construct distributes the execution of the associated region among the threads generated by a `parallel` construct [2]. There is no barrier at the entry. However, if the `nowait` clause is not included, an implied barrier is performed at the end of this construct. This mechanism has been implemented on CAPE. The `cape_barrier()` function is automatically called by the `cape_end()` function to synchronize shared data if no `nowait` clause is declared.

4.5.2.1 `for` construct

The OpenMP `for` construct is designed for C/C++ program. It requires `for` loop to match with the canonical loop form [2]. When reaching a `for` loop construct, the OpenMP program distributes the iterations of `for` loop across the threads. This construct allows adding the schedule, nowait, and data-sharing clauses. In CAPE, the prototype to transform this construct is presented in Fig. 4.8.

```
# pragma omp for [nowait | other-clauses]
for (A ; B ; C)
        D ;
```

↓ is translated to ↓

```
cape_begin(FOR | FOR_NOWAIT, A, B);
        [cape_clause_functions]
        [ckpt_start();]
        for (LEFT ; RIGHT ; C)
                D ;
cape_end(FOR | FOR_NOWAIT, reduction-flag);
```

Figure 4.8: Prototype to transform `pragma omp for`.

`cape_begin()` calculates $LEFT$ and $RIGHT$ of `for` loop in each node. When the `nowait` clause is not indicated, the `ckpt_start()` updates the value of shareable variables after updating their properties which is performed by the relevant clauses. `cape_end()` performs `cape_barrier()` to update the shared data after executing the divided jobs. Otherwise, `ckpt_start()` does not exists and `cape_end()` does nothing. Currently, `dynamic` and `guided` parameters of the `schedule` clause have not been implemented yet.

4.5.2.2 `sections` construct

The OpenMP `sections` construct is a non-iterative work-sharing construct. It contains a set of structured blocks that are distributed and executed by the threads generated by the `parallel` construct. Each structured block is executed once by one of the threads. This construct is translated into CAPE by the prototype presented in Fig. 4.9.

The `cape_section()` function is responsible for identifying which structured block is executed on the current node. At present, structured blocks are indexed and are assigned to nodes so that block i is assigned to node id if id is divides i.

When the `nowait` clause is not indicated, the `cape_start()` function copies the value of shared variables after executing the clause functions. The `cape_barrier()` function is called by `cape_end()` to update shared data among the nodes. Otherwise, `ckpt_start()` does not exist and `cape_end()` does nothing.

```
#pragma omp sections [nowait | other-clauses]
{
        [#pragma omp section]
        D1;
        [#pragma omp section]
        D2;
        ...
}
```

↓ is translated to ↓

```
cape_begin(SECTIONS | SECTIONS_NOWAIT, 0 , 0);
        [cape_clause_functions]
        [ckpt_start();]
        if (cape_section())
                D1;
        if (cape_section())
                D2;
        ...
cape_end(SECTIONS | SECTIONS_NOWAIT, reduction-flag );
```

Figure 4.9: Prototype to transform `pragma omp sections`.

4.5.2.3 `single` construct

The OpenMP `single` construct specifies the associated structured block executed by only one of the threads generated by the `parallel` construct. The executed thread may not be the master one. When a thread is executing, other threads are waiting at a barrier at the end of the `single` construct unless a `nowait` clause is indicated. The translation prototype for this construct is presented in Fig. 4.10.

To implement this mechanism, `cape_single()` is used to identify the first node that enters structured block D. This node is responsible for the execution of D. The others wait for the `cape_barrier()` function if no `nowait` clause is indicated. Otherwise, they continue the execution of the next instructions.

4.5.3 Combined construct

4.5.3.1 `parallel for` construct

The OpenMP `parallel for` construct is a shortcut for specifying a `parallel` construct that contains one `for` construct and no other statement. This construct does not accept the `nowait` clause. Its syntax and translation prototype in CAPE is shown in Fig. 4.11.

At the `cape_begin()` function, the system re-compute the $LEFT$ and the $RIGHT$ values to identify the divided jobs for each node. The `cape_barrier()` function is always executed at `cape_end()` to update shared data after executing the divided jobs.

```
#pragma omp single [nowait | other-clauses]
{
        D;
}
```

↓ is translated to ↓

```
cape_begin(SINGLE | SINGLE_NOWAIT, 0 , 0);
        [cape_clause_functions]
        [ckpt_start();]
        if (cape_single())
                D;
cape_end(SINGLE | SINGLE_NOWAIT, FALSE );
```

Figure 4.10: Prototype to translate `pragma omp sections`.

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

↓ is translated to ↓

```
cape_begin(PARALLEL_FOR, A, B);
        [cape_clause_functions]
        ckpt_start();
        for (LEFT ; RIGHT ; C)
                D ;
cape_end(PARALLEL_FOR, reduction-flag);
```

Figure 4.11: Prototype to translate `pragma omp parallel for`.

4.5.3.2 `parallel sections` construct

The OpenMP `parallel sections` construct is a shortcut for specifying a `parallel` construct that contains one `sections` construct and no other statements. This construct does not accept `nowait` clause. Its syntax and transformation prototype in CAPE is shown in Figure 4.12.

```
#pragma omp parallel sections [clauses]
{
        [#pragma omp section]
        D1;
        [#pragma omp section]
        D2;
        ...
}
```

↓ is translated to ↓

```
cape_begin(PARALLEL_SECTIONS, 0 , 0);
        [cape_clause_functions]
        ckpt_start();
        if (cape_section())
                D1;
        if (cape_section())
                D2;
        ...
cape_end(PARALLEL_SECTIONS, reduction-flag );
```

Figure 4.12: Prototype to translate `pragma omp parallel sections`.

As well as we did for the implementation of `parallel` and `sections` constructs, the `cape_section()` function is called to identify which structured blocks are executed by which nodes. At the `cape_end()` function, `cape_barrier()` is called automatically to update the shared data among the nodes.

4.5.4 Master and Synchronization constructs

4.5.4.1 `master` construct

The OpenMP `master` construct indicates the associated structured block executed by the master thread from the set of threads generated by `parallel` construct. It is no necessary to updated shared data after executing [2]. The translation prototype is presented in Fig. 4.13. In this case, the `cape_master()` function returns $TRUE$ at the master node, and $FALSE$ at the others.

```
#pragma omp master
D ;
```

↓ is translated to ↓

```
if (cape_master())
    D;
```

Figure 4.13: Prototype to translate `pragma omp master`.

4.5.4.2 `critical` construct

The OpenMP `critical` construct specifies the associated structured block executed in one thread at a time. After executing, a barrier is set implicitly to update shared data. The Fig. 4.14 illustrates the translation of this construct. Currently, the order of the execution is set depending on the node index. After execution at each node, `cape_barrier()` is called to update shared data among nodes.

```
#pragma omp critical
D;
```

↓ is translated to ↓

```
for(__i__ = 0; __i__ < cape_get_num_nodes(); __i__++)
{
        if (__i__ == cape_get_node_num())
            D;
        cape_barrier();
}
```

Figure 4.14: Prototype to translate `pragma omp critical`.

4.5.4.3 `flush` construct

The OpenMP `flush` construct executes the `flush()` operation. Hence, when reaching this construct, it is translated into a `cape_flush()` function call.

4.5.4.4 `barrier` construct

The OpenMP `barrier` construct specifies an explicit barrier at the point the construct appears. This construct is translated into a call to the `cape_barrier()` function in CAPE.

4.6 Performance evaluation

In order to evaluate the perforance 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 [92, 93]. These programs have been translated to CAPE and executed on a Cluster to compare the performance. Details of the programs and experimental environment are explained below.

4.6.1 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. The code is shown in Fig. 4.15.

```
int matrix_mult(int A[], int B[], int C[], int n){
        #pragma omp parallel for
        for(i = 0; i < n; i ++){
                for(j = 0; j < n; j++ )
                {
                        for ( k = 0; k < n; k++)
                        {
                                c[i][j] += a[i][k] * b[k][j];
                        }
                }
        }
        return 0;
}
```

Figure 4.15: Multiplication of two square matrices with OpenMP.

2) PRIME: This program counts the number of prime numbers in the range form 1 to N as presented in Fig. 4.16. 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 Equation (4.1).

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \tag{4.1}$$

The OpenMP code is presented in Fig. 4.17. It contains `parallel` and `for` constructs, and data-sharing clauses that is accepted by these construct. This program uses different types of variables in order to test the adaptive granularity implemented on TICKPT.

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. 4.18.

```
int count_prime(int n){
         int i,j, prime, total = 0;
              # pragma omp parallel for shared ( n )
                                        private(j, prime )
                                        reduction(+: total )
              for ( i = 2; i <= n; i++ ){
                     prime = 1;
                     for ( j = 2; j < i; j++ ){
                            if ( i % j == 0 ){
                                   prime = 0;
                                   break;
                            }
                     }
              total = total + prime;
              }
               return total;
}
```

Figure 4.16: The OpenMP code to count the number of prime numbers from 1 to N.

```
double pi(int NUM_STEPS){
        double x , aux, sum, pi, step;
        int i;
        x = 0;
        sum = 0.0;
        step = 1.0 / (double) NUM_STEPS;
        #pragma omp parallel private(i,x,aux) shared(sum)
        {
                #pragma omp for reduction(+: sum)
                for(i=0; i< NUM_STEPS; i++){
                        x = (i+ 0.5) * step;
                        aux = 4.0/(1.0 + x*x);
                        sum += aux;
                }
        }
        pi = step * sum;
        return pi;
}
```

Figure 4.17: The OpenMP function to compute the value of PI.

```
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;
}
```

Figure 4.18: OpenMP function to compute vectors using `sections` construct.

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. 4.19.

```
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 4.19: OpenMP function to compute vectors using `for` construct.

4.6.2 Evaluation context

The experiments have been 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 by Ubuntu 14.03 LTS operation system with OpenSSH-Server and MPICH-2. They are interconnected by a 100Mbps LAN network.

To evaluate the performance of CAPE based on TICKPT and the new execution model (CAPE-TICKPT), we translated and ran the programs presented Sec. 4.6.1. Each experiment has been performed at least 10 times to measure the total execution times measure the total execution times and a confidence interval of at least 95% has been always achieved for the measures. The execution time is the mean of the items. These times are compared with those after being performed by the previous version of CAPE – CAPE based on DICKPT (CAPE-DICKPT) and other solutions as translated into MPI.

At present, CAPE-DICKPT only provide prototypes and support for the `omp parallel for` construct. This version also provide no support for multiple OpenMP directives, OpenMP clauses, and OpenMP programs that do not match with Bernstein's conditions. Therefore, it can only translate and provide the framework to execute the MAMULT2D program.

For the translation to MPI, these OpenMP programs were translated manually into MPI programs using the same behaviors as CAPE. The difference is that, at the synchronization points, CAPE-TICKPT detects the modified data automatically to synchronize while MPI does the same by explicit instruments of the programmers. The translation principle of OpenMP programs into MPI used in this experiment are described as follows:

- The OpenMP `reduction()` clause is translated into the `MPI_Reduce()` function. The other OpenMP clauses are ignored in the MPI program.
- Each `section` directive in `omp sections` is translated to execute in a process in MPI.
- The `iterators` of `omp for` are re-calculated to be assigning for the processes in MPI.
- Shared data are manually identified and sent to relevant processes at the synchronization points.
- The `MPI_SendRecv()` is used to send and receive data between a couple of processes.
- The `MPI_Bcast()` is used to broadcast data from a process to all processes.

4.6.3 Evaluation results

Fig. 4.20 and 4.21 present the execution time in milliseconds for the MAMULT2D program for various size of matrices and different sizes of cluster respectively. Note that, there are many kinds of processors in different nodes. Some of them include many cores, but a single core at each node was used during the experiments. Three measures are presented at each time: the left one (yellow) is associated with CAPE-DICKPT (the previous version), the middle one (blue) is associated with CAPE-TICKPT (the current version), and the right one (red) is associated with MPI.

Fig. 4.20 presents the execution time for various of 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-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. This is due to three major reasons as follows:

- During the computation phase, besides computing the divided jobs, CAPE-DICKPT has to do a series of operations to extract the modified data into an incremental checkpoint, such as set write-protection to all pages of virtual memory, try to write data to write-protection pages, catch `SIGSEGV` signal, copy data of the page, remove write-protection, and re-write the results into that page. While at this phase, on MPI, modified data are identified explicitly by the programmer, while CAPE-TICKPT copies the shared variables at the beginning of this phase, and compares old and new values of variables to identify the modified data at the end of the phase. The analysis and evaluation in Sec. 3.4 showed that the DICKPT mechanism takes a large point in the reduction of the performance of the program.
- At the join phase, on CAPE-DICKPT, all slave nodes have to send incremental checkpoints to the master, the master receives, merges them together and then sends the result back to all slave nodes to inject it into their application's memory while MPI

and CAPE-TICKPT are implemented using Ring or Recursive Doubling algorithms (see Sec. 4.3).

- On the execution model of CAPE-DICKPT (see Sec. 2.4.1), only 15 slave nodes are responsible for computing the divided jobs, while the master only waits for the results form the slaves after distributing jobs to them. With MPI and CAPE-TICKPT, all nodes in the system (including the master) have to compute a part of the job.

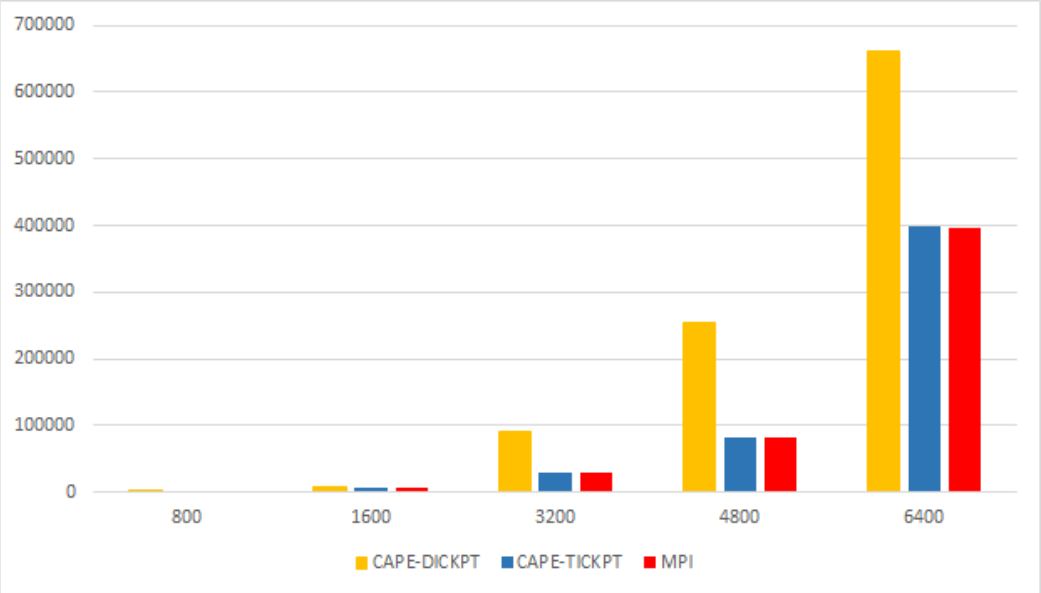

Figure 4.20: Execution time (in milliseconds) of MAMULT2D with different size of matrix on a 16-node cluster.

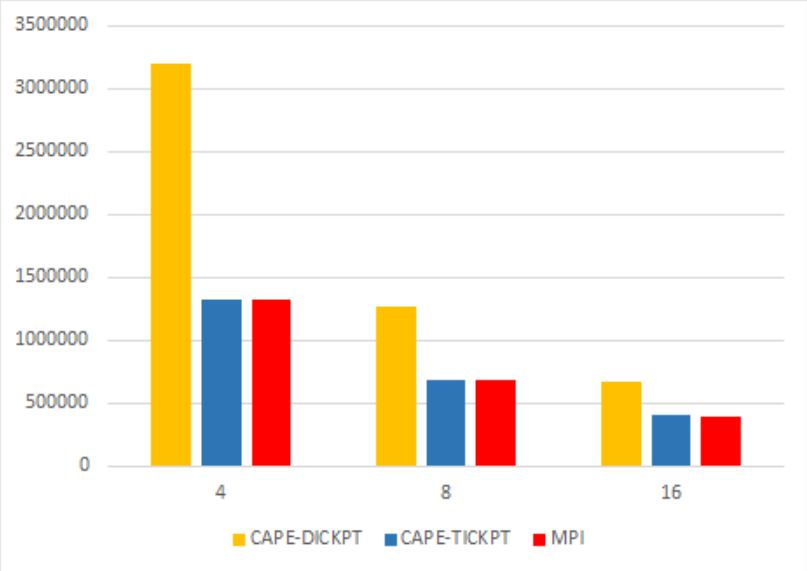

Figure 4.21: Execution time (in milliseconds) of MAMULT2D for different cluster sizes.

Fig. 4.21 presents the execution time for a matrix size of `6400x6400` on different size of a cluster. The 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 CAPE is significantly reduced so that it now much closer to an optimized human-written program using MPI.

To demonstrate the new version of CAPE can run OpenMP programs that do not only match with the Bernstein's conditions while archiving high performance, we did other experiments and compared the performance of CAPE-TICKPT with MPI. All of the four other programs presented in Sec. 4.6.1 have been used to measure the execution time.

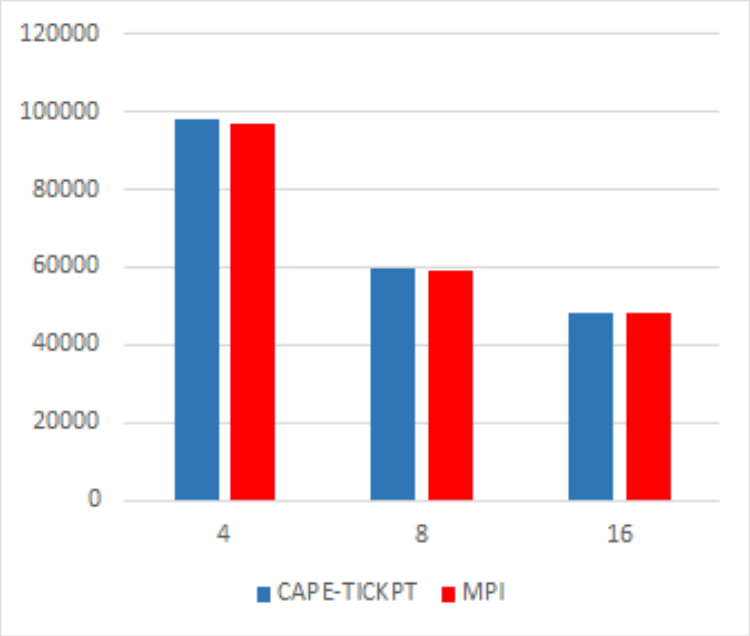

Figure 4.22: Execution time (in milliseconds) of PRIME on different cluster sizes.

Figure 4.22 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 CAPE-TICKPT. In this experiment, the OpenMP `parallel for` directive with the `shared(), private()` and `reduction()` clauses are translated and tested for both two methods. Table 4.2 describes the steps executed by the program for the two methods. The main different step is the join phase. It gathers the results from all nodes and computers the sum of them. The MPI program is clearly specified the values that need to be gathered and calculated the sum, so that the MPI_Reduce() function is called after. CAPE-TICKPT automatically identifies the modified value of the shared variables, extracts 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 considered that we reached to achieve high performance with CAPE.

Fig. 4.23 presents the execution time in milliseconds of PI with $NUM\_STEPS = 10^8$ on 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.

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

Table 4.2: Comparison of the executed steps for the PRIME code for both CAPE-TICKPT and MPI.

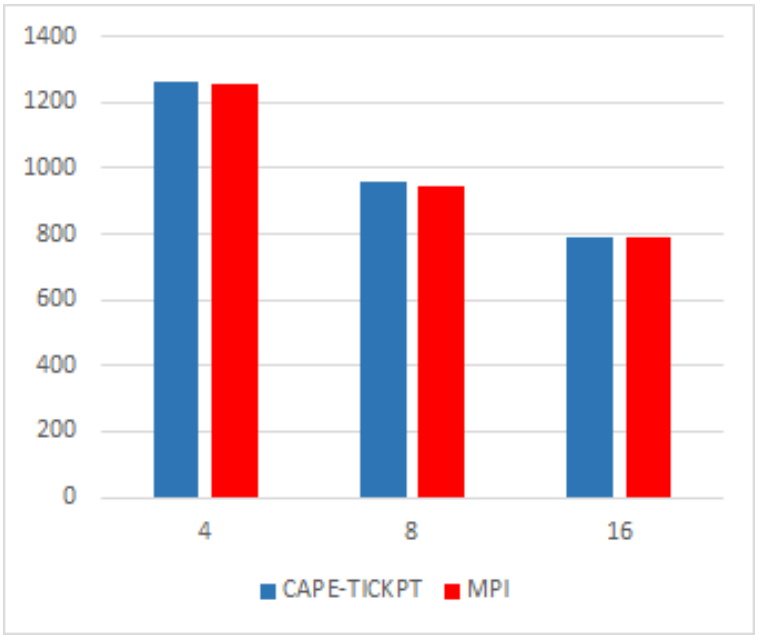

Figure 4.23: Execution time (in milliseconds) of PI on different cluster sizes.

Fig. 4.24 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 on 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. 4.25 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

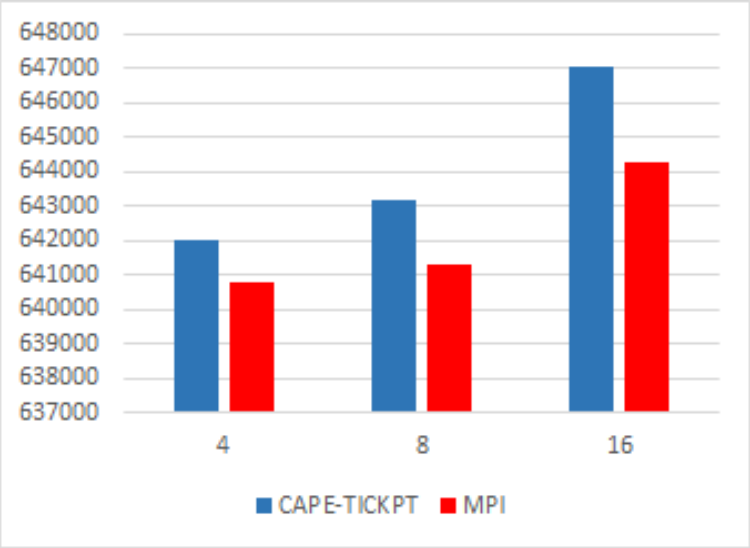

Figure 4.24: Execution time (in milliseconds) of VECTOR-1 on different cluster sizes.

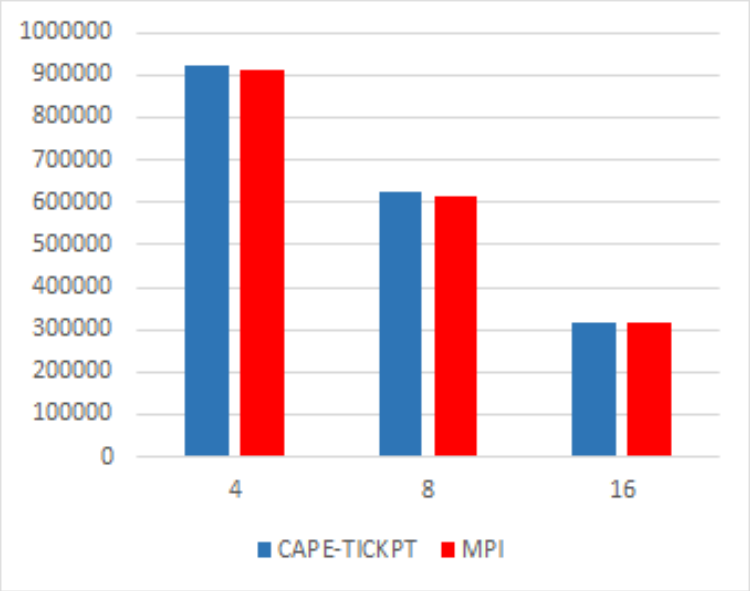

Figure 4.25: Execution time (in milliseconds) of VECTOR-2 different cluster sizes.

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 bock 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.

4.7 Conclusion

This chapter 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 chapter 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 take benefits of the advantages of TICKPT such as checkpoints 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.

Chapter 5

OpenMP Data-Sharing on CAPE

Contents

5.1 Motivation

OpenMP provides a relaxed-consistency shared-memory model [2]. All OpenMP threads can access the same place in the memory to store and retrieve variables. This memory is called the shared memory. Besides, each thread is allowed to have its own memory, called the local memory. The local memory provides a temporary view of the shared 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 the local memory to shared memory is only a mandatory requirement at the synchronization points. On the other hand, the local memory allows the threads to store the private variables accessed by the owner only. The shared or private property of variables determines the updating of the shared memory. In OpenMP, it not only has a set of rules to identify the variables property implicitly, but also provides the `threadprivate` construct and a set of clauses that allow the programmers to set property for variables explicitly.

In the previous works presented in [84], CAPE implemented the RC shared-memory model of OpenMP on distributed-memory architectures. Thank to this method, the shared and private variables of the program are stored in the memory of the different nodes. They are independently accessed by the owner. When the program reaches the synchronization points, these data are synchronized using DICKPT. 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 the application memory. This implementation does not only transfer unnecessary data, but also makes the program work incorrectly.

To solve these issues, in the new design and implementation of CAPE based on TICKPT, the modified data of shared variables are selected to synchronize. To do so, the variable attributes are handled followed by the OpenMP data-sharing attribute rules. In addition, checkpoints are merged using the operations presented in Sec. 3.2. In this chapter, we present the method to handle data-sharing attribute for variables in CAPE-TICKPT.

5.2 OpenMP Data-Sharing attribute rules

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

5.2.1 Implicit rules

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

5.2.1.1 Data-sharing attribute rules for variables referenced inside a parallel region but outside any construct

The OpenMP data-sharing attributes of variables that are referenced in `parallel` regions, but not in a construct, are determined as follows:

- Variables declared outside any `parallel` regions 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 do not appear 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.

5.2.1.2 Data-sharing attribute rules for variables referenced in a parallel region and in another construct

For constructs located inside a `parallel` construct, the OpenMP data-sharing attributes of variables are inherited from the `parallel` region it belongs 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.

5.2.2 Explicit rules

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

According to [2], `threadprivate` is a declarative directive. It specifies those replicated variables , in which each thread has 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 Fig. 2.3. They have to follow a construct that accepts theses clauses. Table 5.1 summaries what OpenMP data-sharing clauses are accepted by which OpenMP constructs. And the functions of theses clauses are also described in Table 3.2.

	parallel	for	sections	single	parallel for	parallel sections
private	✓	✓	✓	✓	✓	✓
firstprivate	✓	✓	✓	✓	✓	✓
lastprivate		✓	✓		✓	✓
shared	✓				✓	✓
default	✓				✓	✓
copyin	✓				✓	✓
copyprivate				✓		
reduction	✓	✓	✓		✓	✓

Table 5.1: The summary of which OpenMP data-sharing clauses are accepted by which OpenMP constructs.

5.3 Data-Sharing on CAPE-TICKPT

Based on the distributed-memory mechanism, the memory is totally distributed on different nodes. Thanks to this, CAPE can design and implement a RC memory model as similar as the OpenMP one. Here, all variables of the program including shared and private variables are stored at all nodes executing the program. When reaching synchronization points, only the modified values of shared variables are extracted into TICKPT and synchronized between all nodes as illustrated in Fig. 5.1. In order to select and extract only the modified values of the shared variables, CAPE-TICKPT is designed within a set of rules and translated prototypes that implement implicit and explicit attribute rules for variables.

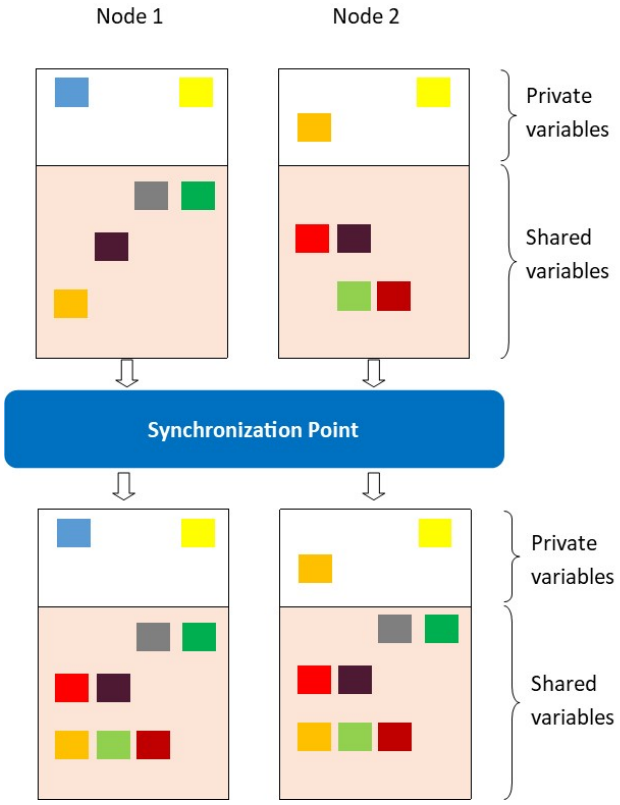

Figure 5.1: Update of shared variables between nodes.

5.3.1 Implementing implicit rules

For detecting the default property of variables, CAPE is designed to follow the OpenMP rules with a small change. It is also added some functions to recognize the variables. Below 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 the attribute of the declared variable.

2. For dynamic memory allocation on the heap, the allocation and the destruction of the memory are handled by adding CAPE functions. They add or remove these allocated space from the variable list.

3. Variables declared outside `parallel` regions are shared, unless they are included in a `threadprivate` directive.

4. All variables declared inside a `parallel` construct that are not `static` or `dynamic` variables are private.

5. The data-sharing attribute of variables of the constructs placed inside a `parallel` construct are inherited from `parallel`.

5.3.2 Implementing explicit rules

For the case of explicit rules of data-sharing attribute for variables, OpenMP directives and clauses are translated into the relevant CAPE runtime functions in order to handle their attributes. Fig. 5.2 and 5.3 present the templates used to translate OpenMP `threadprivate` directive and clauses to CAPE code respectively. Here, the OpenMP `threadprivate` directive is translated into a call to the `cape_set_threadprivate()` function, and the OpenMP clauses are translated into the relevant CAPE clause functions. The `cape_set_threadprivate()` function sets the private attribute to the variable at the corresponding address. The CAPE clause functions are implemented with the same behavior as the OpenMP clauses. They are responsible for changing the variable attributes before taking the checkpoints. The CAPE runtime functions associated with OpenMP clauses are presented and described in Table 5.2.

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

↓ is translated to ↓

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

Figure 5.2: Template to translate `pragma omp threadprivate` to CAPE function call.

5.3.3 Generating and merging checkpoints

CAPE-TICKPT performs a series of operations to generate checkpoints containing the modified value of the shared variables after executing a region of the program. Two important steps of these operations are 1) copying data at the beginning and 2) finding different data at the end of region that marks the checkpoint generation. Here, the `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 the current data (data that are read from memory at the time of reaching the synchronization point). The modified data are stored into the checkpoint file. In addition, if the `reduction` property is set to some variables, their data are also saved into the checkpoint regardless their changes.

```
# 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 5.3: Template to translate OpenMP constructs within data-sharing attribute clauses.

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 5.2: CAPE runtime functions associate with their OpenMP clauses.

This is their initial values and it is necessary to perform the indicated operation in the `reduction` clause.

After generating checkpoints, they are sent to partner nodes. These checkpoints are merged together using the `merging` checkpoint operator (see Sec. 3.2.4). The resulting checkpoint after the join phase is injected into the application's memory to update the shared data.

5.4 Analysis and evaluation

The performance evaluation presented in Sec. 4.6 has demonstrated the performance of CAPE-TICKPT are higher than the previous version. Unfortunately, the previous version of CAPE was not able to support the execution of multiple OpenMP directives as well as OpenMP clauses in a program. Therefore, we cannot run OpenMP containing clauses on top of CAPE-DICKPT to compare performance just like we can do it for the checkpoint size.

This section presents a comparison of checkpoint size for CAPE-TICKPT with the current version of CAPE but data-sharing attribute for variables were not implemented. The VECTOR-2 program is executed on a 16-node cluster (see Sec. 4.6 on p. 72) in order to measure and compare the checkpoint size for both methods.

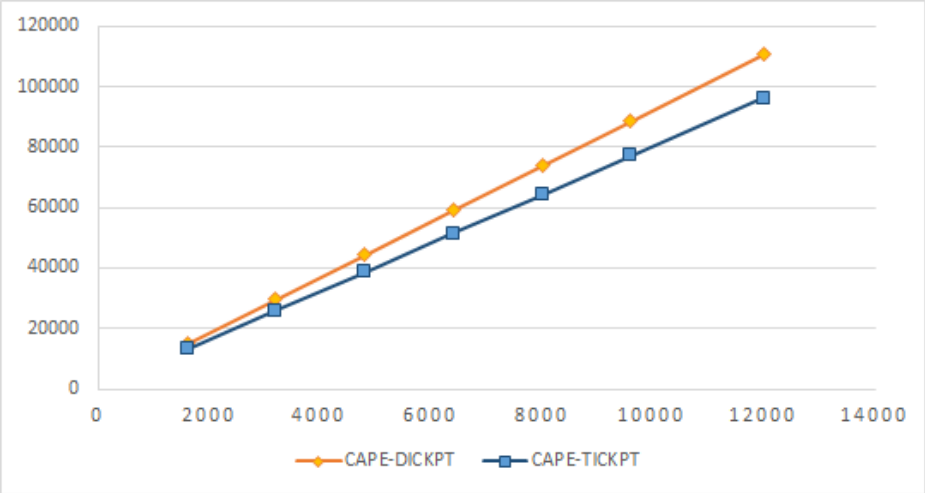

Figure 5.4: Checkpoint size (in bytes) after merging at the master node for both techniques.

Figure 5.4 shows a comparison of the checkpoint size for both methods when executing the VECTOR-2 program on a 16-node cluster with various size of N and M. Here, we assigned $N = M$ and increased the value from 2000 to 14000. According to the figure, the checkpoint size of CAPE-TICKPT is always smaller than the one of CAPE-DICKPT for all measures. This is due to the fact that CAPE-TICKPT only extracts the modified values of shared variables. This means, the modified value of variable i, vector A, and vector Z of the VECTOR-2 program are not saved into the checkpoint file. Unlike CAPE-TICKPT, the previous methods save all modified data, including variables i, j, A, B, Y, and Z.

5.5 Conclusion

This chapter presented the implementation of OpenMP with data-sharing attributes for variables on CAPE. It does not only provide a new capability for CAPE, but also contributes to reduce checkpoint size and improves the performance. The experiments show that the new method makes CAPE more able to reduce the checkpoint size significantly. Moreover, we also found that the programmer can use CAPE efficiently if he/she use OpenMP clauses efficiently to manage data-sharing attributes for variables. We suggest using the `default(none)` clause to set to all variables as private, and then using the `share()` clause to set the sharable property to variables that he/she wants to share among processes.

Chapter 6

Conclusion and Future works

Contents

OpenMP and MPI become the standard tools to develop parallel application on shared and distributed-memory architectures. OpenMP is higher level programming tool than MPI. It is very easy to use. However, it is only developed for shared-memory architectures. CAPE is based on Checkpointing technique had shown the advantages. It translates automatically the OpenMP programs and provides a framework to execute them in parallel on distributed-memory architectures. That help the users to develop parallel application on distributed-memory architecture in easy way.

CAPE have developed and tested over three stages. In the first stage, CAPE is developed based on complete checkpointing. In that, the master node divides jobs and distributes to the slaves using complete checkpoints. The checkpoints size is not optimized in this approach but CAPE have shown that it is a promised approach to port OpenMP on distributed memory architecture while meet fully compliant with OpenMP requirement. In the second stage, CAPE is designed and developed based on Discontinuous Incremental Checkpointing (a kind of Incremental Checkpointing). To execute, the program is initialized and executed at all nodes in the system. The checkpoint technique only take the modified of data. Therefore, the execution time is improved significantly. In the third stage, the checkpoint techniques and execution model of CAPE is optimized. The works presented in this thesis have successfully and significantly improved the ability, reliability, and performance of CAPE.

6.1 Contribution of the thesis

In summary, the main contributions of the thesis are:

- We proposed arithmetics on checkpoints, that defines the data structures and operations to compute directly on checkpoints. From the arithmetics on checkpoints, the checkpoints of a program executed in parallel on different processors can be combined altogether and do not require a parallel program matching with Bernstein's conditions

anymore, which improved the merging time significantly, and improved the reliability of program too.

The checkpoint's data structure contains two main components: register values and memory members. Each component have its own time-stamp in order to record the order when they are taken. The checkpoint's operations are defined, such as *replacement* of memory elements, *merging* and *excluding* checkpoints, etc. that allow merge or find the difference of checkpoints more efficiency. The analysis showed that the execution time of performing these operations are reduce significantly.

- We developed Time-stamp Incremental Checkpoint (TICKPT), a new checkpoint technique as a case study for arithmetics on checkpoints. TICKPT proved that the size can be significantly reduced as compared to DICKPT. In addition, a comparison of the bad affect on the execution time of programs using checkpointing, TICKPT affects performance much less than DICKPT (see Sec. 3.3 and 3.4 on p. 48 and 51).

  To develop TICKPT, we also have implemented two different methods 1) page write-protection and 2) duplication of shared variables to compare and select the best one. The experiment results showed that the method 2) is better than method 1) in the term of checkpointing time and checkpoint's size.

- We designed and implemented a new execution model and new prototypes for CAPE based on TICKPT. This has improved the performance and ability of CAPE. The experiments shown that CAPE can execute OpenMP programs that do not match with the Bernstein's conditions. Moreover, the performance are equivalent to the ones of MPI and are much higher than the previous versions (see Chap. 4 on p. 59).

  In the new execution model of CAPE, when reaching the parallel region, a CAPE functions set up an environment in order to detect the modified data of shared variables. The modification is extracted and stored into TICKPT at the synchronization points. The TICKPTs are automatically synchronized among nodes in the system.

- We implemented OpenMP data-sharing attributes on CAPE. The implementation of OpenMP data-sharing attributes for variables have reduced the checkpoint size generated at the join phase, and contributed to improve the global performance. In particular, it increases CAPE's ability as well as reliability (see Chap. 5 on p. 83).

6.2 Future Works

In the near future, we would like to develop CAPE more complete to distribute an open source solution. To do that, firstly, we have to develop more completely the translation tool that translates from OpenMP source to CAPE source code. That can adapt any kind of OpenMP program. Then, this translation tool is integrate in to a C/C++ compiler. In which, the code can be easier to analyze, re-organize and optimize that can improve the performance of CAPE.

In addition, today, computers are increasingly equipped with more CPUs, GPUs and other accelerators. Then, CAPE have to develop to use efficiently the resource and improve the performance. There are two directions that can be continued to develop CAPE:

- Develop CAPE to support multi-cores. The are two approaches: 1) re-call OpenMP on each node when it executes the divided jobs, or 2) develop checkpoint technique on

CAPE for multi-cores. For method 1), it is easy to develop, we can keep OpenMP code and add some CAPE runtime functions as well as the previous implementation. The new execution model of this approach likes the illustration at Figure 6.1. For method 2), the checkpoint techniques on CAPE have to improve to adapt with execution on multi-cores and avoid the conflict of memory.

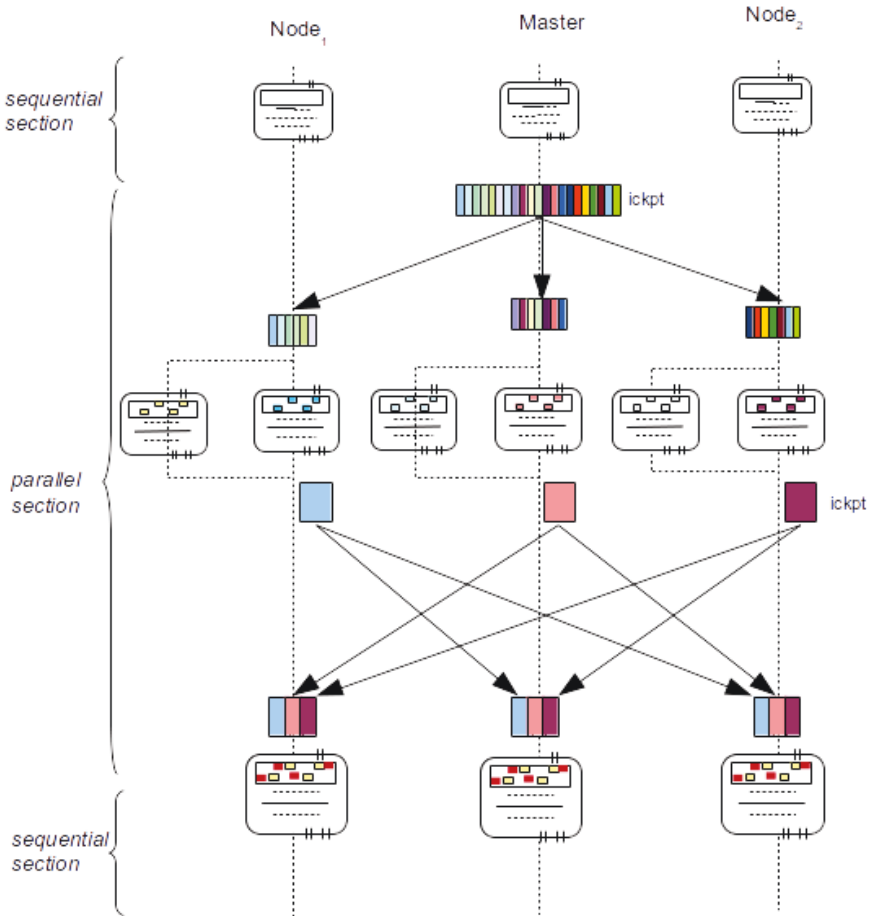

Figure 6.1: The execution model for CAPE while using multi-cores.

- Develop CAPE to support accelerators. In this direction, we will develop checkpoint techniques for CAPE on accelerators. The checkpoints from the accelerators of different nodes should be developed to merge and resume execution portability.

Further more, an another plan is improving the CAPE framework to execute over computers connected through the Internet. Each computer having CAPE framework installed can join the community to compute a part of a parallel job if possible. The master node manages the resources, divides jobs and extracts the checkpoints, distributed to the slave nodes, and handle the returned results. This direction can take the advantages of checkpoint techniques. In that, a part of program code can be saved, sent to another machine, and resumed the execution.

Appendix A

A.1 MPI programs

To compare CAPE with MPI, we manually translated OpenMP source code to MPI source code. This section presents the translated source code in MPI for the benchmark presented in Sec. 4.6.1 (p. 72) .

A.1.1 MAMULT2D

```
int mpi_mamult2d(int A[], int B[], int C[], int n){
        int numtasks, taskid, i, j, k;
        // 1) divide jobs
        int left = taskid * n /numtasks ;
        int right = (taskid + 1) * n / numtasks ;
        // 2) compute divided jobs
        for(i = left; i < right ; i ++)
                for(j = 0; j < N; j++ )
                        for ( k = 0; k < n; k++)
                                C[i][j] += A[i][k] * B[k][j];
        // 3) syncronize the result
        int nsteps = log2 (numtasks);
        unsigned long msg_size = (n * n )/ numtasks ;
        int partner, nleft, nright;
        for(i = 0; i < nsteps; i ++){
                partner = taskid ^ (1 << i);
                MPI_Sendrecv(&left, 1, MPI_INT, partner, i,
                        &nleft, 1, MPI_INT, partner, i,
                        MPI_COMM_WORLD, &status);
                MPI_Sendrecv(&C[left][0], msg_size, MPI_INT, partner, i,
                        &C[nleft][0], msg_size, MPI_INT, partner, i,
                        MPI_COMM_WORLD, &status);
                left = (left <= nleft) ? left : nleft ;
                msg_size = msg_size * 2 ;
        }
        return 0;
}
```

A.1.2 PRIME

```
int mpi_prime(int n, numnodes, nodeid)
{
        int i, j prime, total = 0, result = 0 ;
        // 1) divide jobs
        int s = 2; // This value will be set at the compile time
        int left = nodeid * (n - s) / numnodes + s;
        int right = (nodeid + 1) * (n - s) / numnodes + s;
        // 2) Compute the divided jobs
        for ( i = left; i <= right; i++ ){
                prime = 1;
                for ( j = 2; j < i; j++ ){
                        if ( i % j == 0 ){
                                prime = 0;
                                break;
                        }
                }
                total = total + prime;
        }
        //3) Synchronize the result
        MPI_Reduce(&total, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Bcast(&result, 1, MPI_INT, 0, MPI_COMM_WORLD);
        return 0;
}
```

A.1.3 PI

```
int mpi_pi (int number_of_steps, int numnodes, int nodeid)
{
        double x = 0.0 , aux=0.0 , sum=0.0, result=0.0, pi=0.0;
        double step;
        int i;
        step = 1.0 / (double) number_of_steps;
        //1) divide jobs
        unsigned int n = number_of_steps;
        unsigned int s = 0; // This value will be set at the compile time
        unsigned int left = nodeid * (n - s) /numnodes + s;
        unsigned int right = (nodeid + 1) * (n - s) / numnodes + s;
        //2) compute divided jobs
        for(i = left ; i < right ; i++){
                x = (i+ 0.5) * step;
                aux = 4.0/(1.0 + x*x);
                sum += aux;
        }
        //3) Synchronize the results
        MPI_Reduce(&sum, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
        MPI_Bcast(&result, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        return 0;
}
```

A.1.4 VECTOR-1

```
float a[], float b[], float c[];
//...
int mpi_vector1 (int n, int numnodes, int nodeid)
{
        int i, tid, nthreads;
        tid = nodeid;
        if (tid == 0){
                nthreads = numnodes;
        }
        // node 0: execute block 1
        if (tid == 0 ){
                for (i=0; i<n; i++){
                        for (j= 0 ; j< n; j+=25)
                                b[j] = b[j] * 0.15 ;
                        c[i] = a[i] + b[i];
                }
        }
        // node 1: execute block 2
        if(tid==1){
                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];
                }
        }
        // Synchronize results
        MPI_Bcast(&c, n, MPI_FLOAT, 0, MPI_COMM_WORLD) ;
        MPI_Bcast(&d, n, MPI_FLOAT, 1, MPI_COMM_WORLD) ;
        return 0;
}
```

A.1.5 VECTOR-2

```
float a[N], b [N], y[M], z[M];
\\...
int mpi_vector2 (int n, int m, int numnnodes, int nodeid){
        int i,j;
        //1) devide jobs
        unsigned int s_n = 0;
        unsigned int left_n = nodeid * (n - s_n) /numnodes + s_n;
```

```
        unsigned int right_n = (nodeid + 1) * (n - s_n) / numnodes + s_n;
        unsigned int s_m = 0;
        unsigned int left_m = nodeid * (m - s_m) /numnodes + s_m;
        unsigned int right_m = (nodeid + 1) * (m - s_m) / numnodes + s_m;
        //2) compute divided jobs
        for (i=left_n; i<right_n; i++){
               for(j=0; j<n ; j+=20)
                      a[j] = a[j] + 10.25 ;
               b[i] = (a[i] + a[i-1]) / 2.0;
        }
        for (i=left_m; i<right_m; i++){
               for(j=0; j<n ; j+=20)
                      z[j] = z[j] * 0.025 ;
               y[i] = z[i] * i;
        }
        //3) Synchronize data
        int nsteps = mylog2 (numnodes);
        unsigned long msg_size_N = n/ numnodes ;
        unsigned long msg_size_M= m/ numnodes ;
        int partner;
        int nl;
        int ml;
        for(i = 0; i < nsteps; i ++){
               partner = nodeid ^ (1 << i);
               MPI_Sendrecv(&left_n, 1, MPI_INT, partner, i,
                                    &nl, 1, MPI_INT, partner, i,
                                    MPI_COMM_WORLD, &status);
               MPI_Sendrecv(&left_m, 1, MPI_INT, partner, i,
                                    &ml, 1, MPI_INT, partner, i,
                                    MPI_COMM_WORLD, &status);
               MPI_Sendrecv(&b[left_n], msg_size_N*4,MPI_CHAR, partner,i,
                                    &b[nl], msg_size_N*4,MPI_CHAR, partner,
                                        i,
                                    MPI_COMM_WORLD, &status);
               MPI_Sendrecv(&y[left_m], msg_size_M*4, MPI_CHAR, partner,i,
                                    &y[ml], msg_size_M *4,MPI_CHAR, partner
                                        ,i,
                                    MPI_COMM_WORLD, &status);
               left_n = (left_n <= nl) ? left_n : nl ;
               left_m = (left_m <= ml) ? left_m : ml ;
               msg_size_N = msg_size_N * 2 ;
               msg_size_M = msg_size_M * 2 ;
        }
        return 0;
}
```

A.2 Translation tool for CAPE

In order to translate form OpenMP code to CAPE code, we used a tool implemented by Patricia Reinoso, a MSc. students at Télécom Sudparis. This tool is implemented based on TXL language [94] and the CAPE prototypes presented in Sec. 4.5 (p. 65).

Bibliography

[1] MPI Forum, "Message passing interface forum."

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

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

[4] V. H. Ha and E. Renault, "Design and performance analysis of CAPE based on discontinuous incremental checkpoints," in *2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2011.

[5] ——, "Improving performance of CAPE using discontinuous incremental checkpointing," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE, 2011, pp. 802–807.

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

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

[8] 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, 2002, pp. 195–206.

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

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

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

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

[13] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

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

[15] V. L. Tran, E. Renault, and V. H. Ha, "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, 2016, pp. 620–627.

[16] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[17] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[18] J. J. Costa, T. Cortes, X. Martorell, E. Ayguadé, and J. Labarta, "Running openmp applications efficiently on an everything-shared sdsm," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 35.

[19] A. Basumallik, S.-J. Min, and R. Eigenmann, "Towards openmp execution on software distributed shared memory systems," in *International Symposium on High Performance Computing*. Springer, 2002, pp. 457–468.

[20] Z. Radovic and E. Hagersten, "Removing the overhead from software-based shared memory," in *Supercomputing, ACM/IEEE 2001 Conference*. IEEE, 2001, pp. 9–9.

[21] H. Matsuba and Y. Ishikawa, "Openmp on the fdsm software distributed shared memory," in *The Fifth European Workshop on OpenMP, EWOMP*, vol. 3, 2003.

[22] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi, "Dynamic home node reallocation on software distributed shared memory," in *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, vol. 1. IEEE, 2000, pp. 158–163.

[23] Y. Ojima, M. Sato, H. Harada, and Y. Ishikawa, "Performance of cluster-enabled openmp for the scash software distributed shared memory system," in *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*. IEEE, 2003, pp. 450–456.

[24] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka, "Design of openmp compiler for an smp cluster," in *Proc. of the 1st European Workshop on OpenMP*, 1999, pp. 32–39.

[25] Y.-S. Kee, J.-S. Kim, and S. Ha, "Parade: An OpenMP programming environment for smp cluster systems," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. ACM, 2003, p. 6.

[26] L. Iftode, *Home based shared virtual momory.* Citeseer, 1998.

[27] L. Whately, R. Pinto, M. Rangarajan, L. Iftode, R. Bianchini, and C. L. Amorim, "Adaptive techniques for home-based software dsms," in *Proceedings of the 13th Symposium on Computer Architecture and High-Performance Computing.* Citeseer, 2001, pp. 164–171.

[28] F. Mueller, "Distributed shared-memory threads: Dsm-threads," in *Workshop on Run-Time Systems for Parallel Programming.* Citeseer, 1997, pp. 31–40.

[29] M. Pizka and C. Rehn, "Murks-a posix threads based dsm system," *PDCS'01*, pp. 642–648, 2001.

[30] Y.-S. Kee, J.-S. Kim, and W.-C. Jeun, "Atomic page update methods for openmp-aware software dsm," in *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on.* IEEE, 2004, pp. 144–151.

[31] X. Martorell, J. Labarta, N. Navarro, and E. Ayguadé, "A library implementation of the nano-threads programming model," in *European Conference on Parallel Processing.* Springer, 1996, pp. 644–649.

[32] M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro, and J. Oliver, "Nanoscompiler: supporting flexible multilevel parallelism exploitation in openmp," *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1205–1218, 2000.

[33] H. Kopetz and J. Reisinger, "The non-blocking write protocol nbw: A solution to a real-time synchronization problem," in *Real-Time Systems Symposium, 1993., Proceedings.* IEEE, 1993, pp. 131–137.

[34] R. Buyya, T. Cortes, and H. Jin, "Single system image," *The International Journal of High Performance Computing Applications*, vol. 15, no. 2, pp. 124–135, 2001.

[35] INRIA, "Kerrighed," 2010. [Online]. Available: http://www.kerrighed.org

[36] C. Morin, P. Gallard, R. Lottiaux, and G. Vallée, "Towards an efficient single system image cluster operating system," *Future Generation Computer Systems*, vol. 20, no. 4, pp. 505–521, 2004.

[37] R. Lottiaux and C. Morin, "Containers: A sound basis for a true single system image," in *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on.* IEEE, 2001, pp. 66–73.

[38] J. Breitbart, "Analysis of a memory bandwidth limited scenario for numa and gpu systems," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on.* IEEE, 2011, pp. 693–699.

[39] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff, "A hybrid approach of OpenMP for clusters," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 75–84.

[40] A. J. Dorta, J. A. Gonzalez, C. Rodriguez, and F. De Sande, "llc: A parallel skeletal language," *Parallel Processing Letters*, vol. 13, no. 03, pp. 437–448, 2003.

[41] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for epex/fortran," *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.

[42] S.-J. Min, A. Basumallik, and R. Eigenmann, "Optimizing openmp programs on software distributed shared memory systems," *International Journal of Parallel Programming*, vol. 31, no. 3, pp. 225–249, 2003.

[43] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, 1991.

[44] Y. Lin and D. Padua, "Compiler analysis of irregular memory accesses," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 157–168, 2000.

[45] J. Hoeflinger, "Intel cluster openmp," 2010. [Online]. Available: https://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers

[46] C. Terboven, D. An Mey, D. Schmidl, and M. Wagner, "First experiences with intel cluster openmp," in *International Workshop on OpenMP*. Springer, 2008, pp. 48–59.

[47] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A portable" shared-memory" programming model for distributed memory computers," in *Supercomputing'94., Proceedings*. IEEE, 1994, pp. 340–349.

[48] ——, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[49] L. Huang, B. Chapman, Z. Liu, and R. Kendall, "Efficient translation of openmp to distributed memory," in *International Conference on Computational Science*. Springer, 2004, pp. 408–413.

[50] L. Huang, B. Chapman, and R. Kendall, "OpenMP for clusters," in *The Fifth European Workshop on OpenMP, EWOMP*, vol. 3, 2003.

[51] L. Huang, B. Chapman, and R. A. Kendall, "Openmp on distributed memory via global arrays," in *Advances in Parallel Computing*. Elsevier, 2004, vol. 13, pp. 795–802.

[52] I. Cores, M. Rodríguez, P. González, and M. J. Martín, "Reducing the overhead of an MPI application-level migration approach," *Parallel Computing*, vol. 54, pp. 72–82, 2016.

[53] V. L. Tran, E. Renault, and V. H. Ha, "Improving the reliability and the performance of CAPE by using MPI for data exchange on network," in *Mobile, Secure, and Programmable Networking: First International Conference, MSPN 2015, Paris, France, June 15-17, 2015, Selected Papers*, vol. 9395. Springer, 2015, p. 90.

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

[55] J. Mehnert-Spahn, E. Feller, and M. Schoettner, "Incremental checkpointing for grids," in *Linux Symposium*, vol. 120. Citeseer, 2009.

[56] I. Cores, G. Rodríguez, P. González, R. R. Osorio *et al.*, "Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes," *New Generation Computing*, vol. 31, no. 3, pp. 163–185, 2013.

[57] V. F. Nicola, *Checkpointing and the modeling of program execution time*. University of Twente, Department of Computer Science and Department of Electrical Engineering, 1994.

[58] R. Badrinath, C. Morin, and G. Vallée, "Checkpointing and recovery of shared memory parallel applications in a cluster," in *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*. IEEE, 2003, pp. 471–477.

[59] O. O. Sudakov, I. S. Meshcheriakov, and Y. V. Boyko, "Chpox: transparent checkpointing system for linux clusters," in *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2007. IDAACS 2007. 4th IEEE Workshop on*. IEEE, 2007, pp. 159–164.

[60] G. Vallée, C. Morin, R. Lottiaux, J. Berthou, and I. D. Malen, "Process migration based on gobelins distributed shared memory," in *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*. IEEE, 2002, pp. 325–325.

[61] C. R. Landau, "The checkpoint mechanism in keykos," in *Object Orientation in Operating Systems, 1992., Proceedings of the Second International Workshop on*. IEEE, 1992, pp. 86–91.

[62] J. Ansel, K. Aryay, and G. Coopermany, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[63] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette *et al.*, "Mpich-v: Toward a scalable fault tolerant MPI for volatile nodes," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 29–29.

[64] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of MPI programs," *ACM Sigplan Notices*, vol. 38, no. 10, pp. 84–94, 2003.

[65] G.-M. Chiu and J.-F. Chiu, "A new diskless checkpointing approach for multiple processor failures," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 4, pp. 481–493, 2011.

[66] A. Beguelin, E. Seligman, and P. Stephan, "Application level fault tolerance in heterogeneous networks of workstations," *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 147–155, 1997.

[67] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke, "Portable checkpointing and recovery," in *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*. IEEE, 1995, pp. 188–195.

[68] D. Cummings and L. Alkalaj, "Checkpoint/rollback in a distributed system using coarse-grained dataflow," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on.* IEEE, 1994, pp. 424–433.

[69] C.-C. Li and W. K. Fuchs, "Catch-compiler-assisted techniques for checkpointing," in *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium.* IEEE, 1990, pp. 74–81.

[70] Z. Chen, J. Sun, and H. Chen, "Optimizing checkpoint restart with data deduplication," *Scientific Programming*, vol. 2016, 2016.

[71] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *European Conference on Parallel Processing.* Springer, 2011, pp. 302–311.

[72] T. A. Welch, "Technique for high-performance data compression," *Computer*, no. 52, 1984.

[73] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication." in *FAST*, 2013, pp. 183–198.

[74] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory compression: coarse-grained data reordering to improve compressibility." in *FAST*, 2014, pp. 257–271.

[75] J. Heo, S. Yi, Y. Cho, J. Hong, and S. Y. Shin, "Space-efficient page-level incremental checkpointing," in *Proceedings of the 2005 ACM symposium on Applied computing.* ACM, 2005, pp. 1558–1562.

[76] J. S. Plank, J. Xu, and R. H. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," Citeseer, Tech. Rep., 1995.

[77] N. Hyochang, K. Jong, S. J. Hong, and L. Sunggu, "Probabilistic checkpointing," *IEICE TRANSACTIONS on Information and Systems*, vol. 85, no. 7, pp. 1093–1104, 2002.

[78] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing different approaches for incremental checkpointing: The showdown," in *Linux'11: The 13th Annual Linux Symposium*, 2011, pp. 69–79.

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

[80] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography.* CRC press, 1996.

[81] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proceedings of the 18th annual international conference on Supercomputing.* ACM, 2004, pp. 277–286.

[82] F. A. F. B. Gomes, *Optimizing incremental state-saving and restoration.* University of Calgary, 1996.

[83] D. West and K. Panesar, "Automatic incremental state saving," in *ACM SIGSIM Simulation Digest*, vol. 26, no. 1. IEEE Computer Society, 1996, pp. 78–85.

[84] V. H. Ha, "Optimisation de la gestion mémoire sur machines distribuées," Ph.D. dissertation, Informatique, Télécommunications et Electronique de Paris, Télécom SudParis, 10 2012.

[85] P. J. Salzman, M. Burian, and O. Pomerantz, "The linux kernel module programming guide," 2007. [Online]. Available: https://www.tldp.org/LDP/lkmpg/2.6/html/index.html

[86] L. Mereuta and É. Renault, "Checkpointing aided parallel execution model and analysis," in *High Performance Computing and Communications*. Springer, 2007, pp. 707–717.

[87] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Transactions on Electronic Computers*, no. 5, pp. 757–763, 1966.

[88] V. Alfred, S. L. Monica, S. Ravi, and D. U. Jeffrey, *Compilers Principles, Techniques,& Tools*. Addion Wesley, 2006, vol. 2.

[89] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[90] NASA Advanced Suppercomputing Division, "Nas parallel benchmarks." [Online]. Available: https://www.nas.nasa.gov/publications/npb.html

[91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[92] J. M. Bull and D. O'Neill, "A microbenchmark suite for openmp 2.0," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 5, pp. 41–48, 2001.

[93] EPCC, "Epcc openmp micro-benchmark suite." [Online]. Available: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite

[94] H.-H. Charles and C. James, "The txl programming language," May 2017. [Online]. Available: https://www.txl.ca