2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

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

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

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

***Abstract*—CAPE (Checkpointing-Aide Parallel Execution) is an approach tried to port OpenMP programs on distributed memory architectures like Cluster, Grid or Cloud systems. It provides a set of prototypes and functions to translate automatically and execute OpenMP program on distributed memory systems based on the checkpointing techniques. This solution has shown that it has achieved high performance and complete compatibility with OpenMP. However, it is in research and development stage, so there are many functions that need to be added, some techniques and models need to be improved.**

This paper presents approaches and techniques that have been applied and will be applied to optimize checkpoints and execution model of CAPE.

***Index Terms*—CAPE; Checkpointing-Aide Parallel Execution; OpenMP; HPC; parallel computing;**

I. Introduction

In order to minimize programmers' difficulties when developing parallel applications, a parallel programming tool at a higher level should be as easy-to-use as possible. MPI [1] 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, request the programmer to tag the pieces of the code to execute in parallel.

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

CAPE adopts a completely different approach based on the use of the checkpointing techniques [9] [10] [11] [12] [13] [14]. This allows distributing the work of the parallel block programs to the different processes of the system, and to handle the exchange of shared data automatically. Two versions of CAPE [15] [13] have been released, which proves the effectiveness of the approach and the high-performance.

In spite of under development, CAPE has shown that it is a possible solution. In some comparison with MPI, CAPE reaches around 90% performance [12]. In addition, CAPE is fully compatible with OpenMP [12] [13]. However, in order to become more completed, some new models need to be suggested and tested, some technical using in CAPE need to be improved.

In this subject, we will optimize checkpointing and CAPE's execution model, which will improve the performance, ability, capability of CAPE.

II. CAPE principles

At this moment, two versions of CAPE have been developed. The first one using complete checkpoints [15] [16] and the second one using incremental checkpoints [12] [13]. The second one is named CAPE-2 that demonstrated high performance and complete compatibility with OpenMP [13].

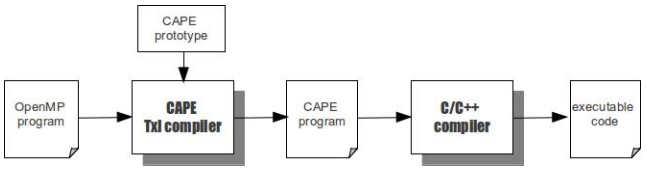

Fig. 1. Translation OpenMP programs with CAPE

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

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.119

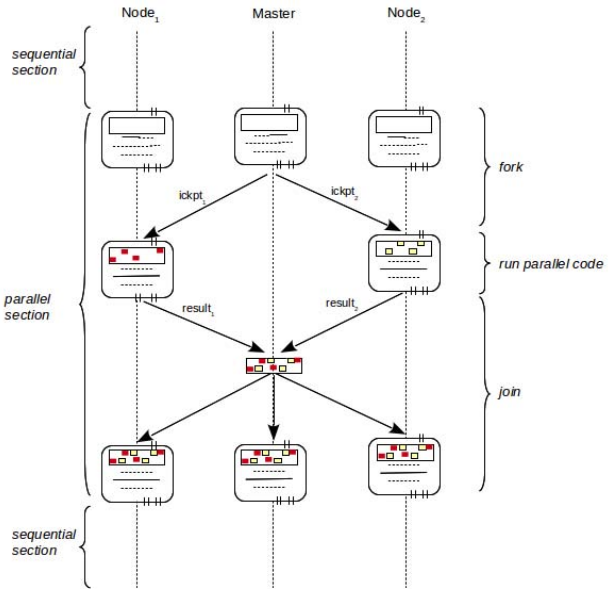

Fig. 2. CAPE execution model

on distributed-memory systems, and discuss transformation of prototypes in the next part of this section.

A. Execution model

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

In order to execute a CAPE code on distributed-memory architectures, the program first run on set of nodes on the system, each node works as a process. Whenever the program meets a parallel section, the master distributes jobs to the slaves process by using Discontinuous Incremental Checkpoints (DICKPT) [12] [17]. By this approach, the master node generates DICKPTs and sends them to slave nodes, each slave node will receive a checkpoint. After sending checkpoints, the master will wait for the returned results from slaves. The next step of master is that they receive and merge the result of the checkpoints before injecting this into its memory. For slave nodes, they receive different checkpoints, and then, they inject this checkpoint into their memory to do the divided job. The result will be sent back to master by using DICKPTs. The end of the parallel region, the master sends the result of the checkpoint to every slaves to synchronize the memory space of the program.

B. Transformations from OpenMP to CAPE source code

In CAPE framework, we defined and implemented a set of functions that perform tasks such as generate, distribute, receive, and inject DICKPTs into memory of program, etc.. Besides, in CAPE compiler, we define a set of prototypes to translate OpenMP program into the form of CAPE programs that can be executed into CAPE framework. At version 2,

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

↓ automatically translated into ↓

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

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

CAPE does not support nested and shared-data variables constructs. Therefore, CAPE just works for the OpenMP programs that match the Bernstein's conditions.

Using the prototypes in CAPE compiler, the OpenMP source code will be translated into CAPE form that does not contain OpenMP directives and structures any more. For more explanation about this mechanism, we will present a prototype for the *parallel for* construct, one of the most complex work-sharing constructs of OpenMP [18]. This will be shown in the Fig. 3.

C. Remarks

In spite of under development, CAPE has shown that it is a possible solution. In some comparisons with MPI, CAPE reaches around 90% performance [12] [19]. In addition, CAPE is fully compatible with OpenMP [12] [13]. However, in order to become more completed, some new models need to be suggested and tested, some techniques using in CAPE need to be improved. In this section, we present some remarks, analyse some weaknesses of the CAPE. They will be solved in this subject.

- Firstly, as shown in the Fig. 2, there is the risk of bottlenecks in the master node, especially in the step waiting for checkpoints from slaves, merge checkpoint, and send slaves back to synchronize program memory.
- Secondly, in order to distribute jobs to slaves, the master generates a number of checkpoints depending on the number of slave nodes so that each slave nodes will receive a checkpoint. This method achieves maximum optimization in this step. However, it is not flexible because of three following reasons: 1) The number of slaves should not be identified at compile step; 2) OpenMP source code should be modified to detect when the master generates checkpoint; 3) The dynamic scheduling of OpenMP can not be implemented by this method.
- Thirdly, after distributing jobs, the slaves will execute the divided jobs while the master does not do anything until receiving checkpoint's result from slaves. That wastes the resources.
- Fourthly, at each node in the cluster, today, the computer processing is equipped more than two cores. Therefore, the performance will be improved significantly if we can use multi-cores architecture of each node in the system.
- Fifthly, OpenMP shared-data variables environment is an important element of OpenMP, it needs to be supported on CAPE.

III. Checkpoint Optimization

A. Checkpointing techniques

Checkpointing is the technique that tries to save image of the state of processes at a point during its lifetime, and then, it can be resumed at the saving's time if necessary [9] [14]. Processes can resume execution from a checkpoint state when a failure occurs, therefore, no need takes time to initialize and execute it from the beginning. These techniques have been introduced since two decades ago. Nowadays, they are researched and used widely on on fault-tolerance, applications of trace/debugging, roll-back/animated playback, and process migration.

In order to optimize the size of checkpoint, to reduce the time and overhead of checkpointing, there are many techniques which have been introduced. In those, incremental checkpoint [9] [10] [11] [20] [21] is the best technique. The main idea of incremental checkpoint is just saving the change of the state of program compared with the previous checkpoint.

For incremental checkpointing, detecting modified memory can be based on page-fault or hash-table mechanism. The granularity can be a page or a word. Besides, the researches applied some another techniques to reduce checkpoint size such as Memory exclusion [9], Checkpoint compression [9] [10], User-directed checkpointing [9], combination of page protection and hash table [21], etc..

B. Checkpoint optimization on CAPE

On CAPE, incremental checkpointing is a very important factor that saves a part of state of program in a process, and then resumes at another process. This technique is used to distribute jobs and synchronize data automatically.

In our works, we continued to use incremental checkpoint with page protection mechanism, and detected the modified memory by setting the granularity of the checkpoint to a word. Furthermore, in an another research [17], we have shown that, in order to take advantage of the spatial locality of updates and reduce the size of checkpoints, several alternative methods for storing memory updates have been identified. They are Single data (SD) - data which will be stored as a pair of address-value of a word; several successive data (SSD) - data which will be saved as a triple address, size, and value; many data (MD) - save data by address, a map, and value; Entry page (EP) - save data by address, value. Depending on the structures of checkpoint and the number of updates, the size of checkpoint is illustrated in Fig. 4.

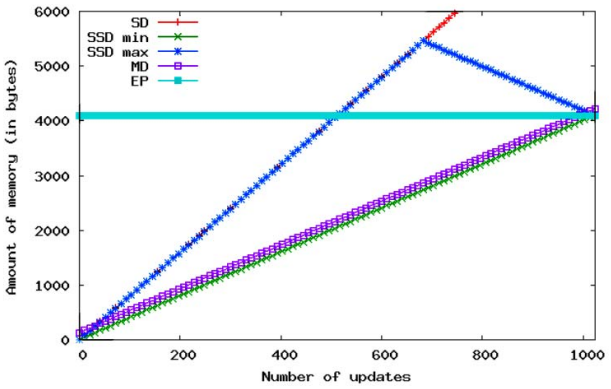

Fig. 4. Amount of memory to store updates

In this section, we used dynamic structure in checkpoint file, depending on the amount of memory modified in a page, the correct structure will be used. A checkpoint file can use multi-structure. This mechanism will reduce checkpoint size.

IV. Design and implementation new execution model for CAPE

In order to solve the weakness mentioned in the section II-C, we have designed and are going to implement the new execution model for CAPE. This model is shown in Fig 5.

To adapt the new model, CAPE prototype is re-designed. For example, the prototype of *parallel for* will be designed as the Fig. 6.

Besides, we built a set of operations acted on checkpoints. This will open a new mechanism allowed us to implement OpenMP's sharing-data attributes. Therefore, CAPE can work for OpenMP program that overcomes the Bernstein's conditions.

V. Conclusions and Future work

A. Conclusions

OpenMP is a very easy-to-use framework to develop parallel application on shared-memory architectures. In the combi-

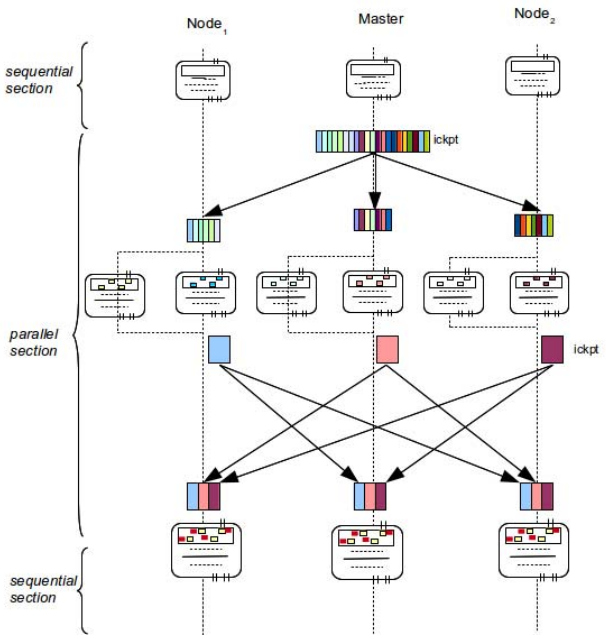

Fig. 5. A new execution model of CAPE

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

↓ automatically translated into ↓

```
1 if ( master ( ) )
2   for (A ; B ; C)
3     generate_dickpt (before_i)
4 scatter(before, &recv_n ,master)
5 inject(recv_n )
6 D
7 generate_dickpt (after_n)
8 allreduce(after_n,&after,[<ops>])
9 inject(after)
```

Fig. 6. Prototype for the *parallel for* for new model.

nation with CAPE, these programs can be easily worked in parallel on distributed memory architectures such as Cluster, Grid and Cloud systems.

We have designed and implemented dynamic structures mechanism for CAPE. We also have built arithmetic mechanism on checkpoint and applied to implement OpenMP's sharing-data attributes clauses and directives. In addition, the new model has been designed and tested.

B. Future work

In order to make CAPE become a completed framework, there are a lot of works to do. Fist of all, we will continue to focus on implementing some important directives of OpenMP on new model to test CAPE by using NAS benchmark. Then, we will develop CAPE by using checkpointing on GPUs.

REFERENCES

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

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

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

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

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

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

[7] 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.

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

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

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

[11] 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.

[12] 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.

[13] ——, "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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.