Improving Performance of CAPE using Discontinuous Incremental Checkpointing

Viet Hai Ha, Eric Renault

Improving Performance of CAPE using Discontinuous Incremental Checkpointing

Viet Hai Ha and Éric Renault
Institut Télécom – Télécom SudParis
Samovar UMR INT-CNRS 5157
Évry, France
viet_hai.ha@it-sudparis.eu and eric.renault@it-sudparis.eu

***Abstract*—Originally, OpenMP was designed to develop parallel applications on shared-memory architectures. One of the advantages that made the success of OpenMP is the simplicity of the associated programming model. Checkpointing Aided Parallel Execution (CAPE) is a paradigm that uses checkpointing techniques to run parallel programs on distributed-memory architectures. In order to show its effectiveness, it has been used to develop a compiler to run OpenMP programs on distributed-memory architectures. The first prototype we developed proved the feasibility of the paradigm but the use of complete checkpoints led to poor performance. This was mainly due to the large amount of data to transfer and process. This paper presents the new prototype we developed for CAPE based on the discontinuous incremental checkpointing technique and an analysis its performance.**

I. Introduction

There has been an extensive developement of parallel architecture systems for the last three decades, with the appearance of new concepts like clusters, grids and more recently, multicore processors and clouds. Tools to develop applications on top of these platforms also have evolved but differently. Setting aside objects such as threads and processes, MPI which was designed for distributed-memory systems and OpenMP for shared-memory architectures are the most common standards for parallel programming today. From the programmer's point of view, OpenMP is simpler than MPI as only a set of OpenMP directives need to be inserted to convert a sequential program into a parallel one. OpenMP also provides efficient mechanisms for data sharing and thread synchronization. Its main weakness (from a distributed-memory programer's point of view) is the limitation to shared-memory infrastructures. As a result, OpenMP is convenient for multicore and multiprocessor machines, but for the other systems such as grids, clusters and clouds, a transformation is required. CAPE, our paradigm, may be used to implement this transformation.

This article aims at showing how the performance of the CAPE (Checkpointing Aided Parallel Excecution) paradigm has been improved using discontinuous incremental checkpointing and providing an analysis of these performances. The article is organized as follows: the next two sections present how OpenMP has been ported on different distributed architectures and the principles associated with checkpointing respectively. Sec. IV presents the principles inherent to CAPE and the next section develops how these principles have been modified to cope with discontinuous incremental checkpoints. The last section before the conclusion is dedicated to the analysis of some performance measurements.

II. OpenMP on distributed memory architectures

There have been several attemps to make OpenMP programs running on a distributed-memory architecture [1]. They can be divided into two categories.

The first category consists in using a Single System Image (SSI) in order to hide to the parallel application the distributed nature of the underlying architecture [2]. An SSI aims at providing an abstract layer on top of a distributed system so that users and applications can use resources as if they were parts of a single monolithical machine. Typically, this means that the set of processors is seen as an SMP or a single multicore processor; the set of RAMs available in each node is seen as a single memory; and the set of filesystems are accessible as a single one. Several SSIs are now available and mature with different capabilities and targetting different architectures. Some solutions for clusters are Genesis [3], Millipede [4], Nomad [5] and Kerrighed [6]. More recently, XtreemOS [7], a solution for grid has been developped. XtreemOS is derivated from Kerrighed and still an ongoing work now also targetting clouds [8]. The main advantage of using an SSI to run an OpenMP program is that the program can run as is, with no need to recompile if an executable file is already available.

The second category consists in using a parallel library and translating OpenMP directives and memory updates to calls to the parallel library functions. Two solutions have emerged as of today, one developed on top of MPI [9] and the other one on top of Global Arrays [10]. The main criticism towards solutions based on a parallel library is that they have lots of difficulties to take into account all memory accesses. Typically, when a memory location is accessed through a dereferentiation or a set of dereferentiations, it is sometimes very difficult for the compiler to identify the data type associated with the memory area.

III. Checkpointing

A. Principles and tools

When running, a process may crash at two different levels: hardware or software. Software problems are usually due to a bad use of the language that had not been detected at

compilation time. Hardware problems occur when one of the critical components of the machine fails. If software problems can be tracked using some tools and corrected before a large and long-time running program is started, it is more difficult to track and avoid potential hardware problems. As a result, in order to avoid loosing the result of several hours, days or even weeks of computations, i.e. to insert fault tolerance into computing systems, some systems include a mechanism to regularly save the state of processes, called a checkpoint, so as to be able to resume their execution from the last checkpoint in case of any problem instead of restarting from the beginning of the execution. Typically, a checkpoint consists in saving a snapshot containing the application state, i.e. all the data that will be necessary to restart the process from this state.

The checkpoint may be performed at different levels in the system. *User-level checkpointing* requires no support from the kernel and is usually performed by the process itself. *Kernel-level checkpointing* requires the kernel to extract all the information about the process so that they can be saved on the disk. Both methods have advantages and drawbacks. The main advantage for user-level checkpointing is that quite portable from one system to another without the need of intervention the kernel. The main advantage of kernel-level checkpointing is that all data related to all processes are accessible. Moreover, there exists some hybrid mechanisms where kernel data are accessible from user space. This is the case of checkpointing tools using the `/proc` pseudo-filesystem under Unix.

There are two main categories of checkpointing techniques. *Complete checkpointing* aims at storing all the information related to the process, including the virtual address space, state registers, and sometimes extra information from the kernel like the list of open file descriptor. *Incremental checkpointing* [11] aims at only storing the parts of the memory[1] that have been updated since the beginning of the execution of the program or the last checkpoint. From a high-level point of view, this is perfomed by setting access rights to read-only to all memory pages in order to force the system to deliver a SIGSEGV signal the first time a page is accessed for writing. Upon reception of the SIGSEGV signal, a copy of the content of the page is stored and access rights to the memory pages are restored to their initial values. When a checkpoint is required, the content of all modified pages are compared to their initial content and the difference is stored in the checkpoint file. In order to perform these operations, each process that may be checkpointed has to be associated a monitor. Typically, this monitor is in charge of starting the process which checkpoints will be computed, catching the SIGSEGV signales, generating the checkpoint files and waiting for the terminaison of the process.

Various tools have been developed to checkpoint applications. Some examples are : Libckpt [12], a transparent incremental checkpointing tool running at the user-level; TICK [13], a transparent incremental checkpointing tool too but running at the kernel-level; DMTCP [14] takes into account the specificities of parallel applications including multi-threaded ones and is running at the user level; and CLIP [15] have been designed to take into account the specific case of message-passing libraries.

[1]References to the memory are, unless otherwise specified, references to the virtual memory and not the physical memory. In the same way, the paper refers to virtual pages and virtual address spaces and not physical pages and physical address spaces respectively.

IV. CAPE PRINCIPLES

CAPE, which stands for Checkpointing Aided Parallel Execution, aims at automatically transforming a parallel shared-memory program so that it can be executed on a distributed-memory architecure. In the current implementation we developed, CAPE is able to handle OpenMP directives provided in a C programming language program. However, CAPE is not language dependent and could be extended to any programming language and/or parallel library or tool.

The basic idea of CAPE is that while many researches have been conducted in order to develop checkpointing applications to save the state of a parallel program, CAPE makes use of checkpoints in order to allow programs to run on a distributed-memory architecture instead of a shared-memory architecture. The main difference between these two architectures remains in the fact that two segments out of three (both text and data segments vs. the stack) are shared by the different threads belonging to a single parallel process while the two processes belonging to a single parallel application are executing in two completely different memory address spaces. For both architectures, stacks belonging to different threads are stored at different locations. And as a thread should not access the private data of another directly, there should be no portability problem from a shared-memory architecture to a distributed-memory architecture. As most programs executing on distributed-memory architectures are SPMD, the text segment which is read-only by definition is not changed during execution and is therefore consistent among all the nodes. Thus, this segment involves no problem either. The situation is different for the data segment which memory locations may be accessed by any thread at any time. In the scope of CAPE, the virtual address space is taken into account as a whole and no difference is made among the different segments.

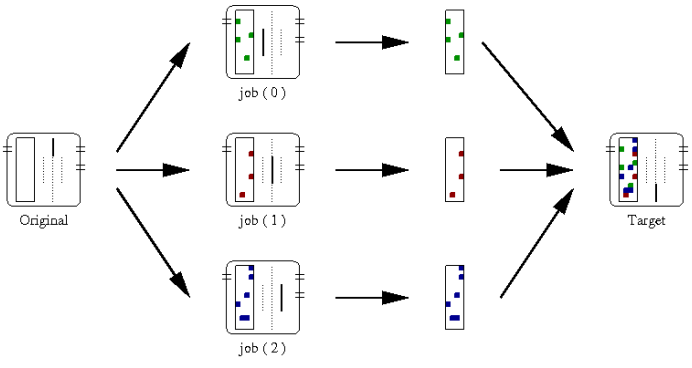

Fig. 1. The basic steps when execution OpenMP programs with CAPE.

Figure 1 presents the main steps that are involved during the execution of a program using CAPE. Assume a program

is composed of three parts. The first and the last parts are composed of a single thread, i.e. they are to be executed sequentially. However, the second part is a parallel part that shall be executed using three threads. The execution is as follows:

1) The program starts using a single thread. Let the node on which this part is executed be the master node.
2) After the execution of the sequential part, a checkpoint is generated and sent to three slave nodes.
3) On each slave node, one thread of the parallel part is executed.
4) After threads have finished their execution on slave nodes, each slave node generates a new checkpoint and computes the difference from the original checkpoint that was previously received on the node, i.e. any memory locations that have been updated is reported.
5) Each slave node sends back the difference to the master node.
6) After all differences from all slave nodes have been received, they are injected in the target process, i.e. the original process that executed the first sequential part.
7) The target process can resume the second sequential part of the program as if the part that have been executed remotely in parallel had been executed locally.

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

↓ automatically translated into ↓

```
parent = create ( original )
if ( parent )
      copy ( original, target )
      ssh host1 restart ( original )
      P1
      parent = create ( after1 )
      if ( parent )
            diff ( original, after1, delta1 )
            wait_for ( delta2 )
            merge ( target, delta2 )
            merge ( target, delta1 )
            restart ( target )
else
      P2
      parent = create ( after2 )
      if ( parent )
            diff ( original, after2, delta2 )
      exit
```

Fig. 2. Template for OpenMP with complete checkpoints.

Figure 2 presents the effective transformation that is performed on a code that specifies parallel regions using OpenMP directives. This transformation uses two parallel sections for the example. However, it can be generalized to an as-large-as-possible set of sections [16] and to loops [17]. In the present example, the parent, i.e. the master node, is in charge of both managing the slave and executing one thread in the parallel part. However, this is not mandatory. Assumes that P_1 and P_2 satify Bernstein's conditions. The translation is based on the following functions:

- `create` create a checkpoint and saves it in the file provided as a parameter. The value returned by the function is used to identify whether the function has just created the checkpoint and returned, or the process has been created after resuming the execution from the checkpoint. This function is very similar to the `fork` system call, except that `create` returns `TRUE` after generating the checkpoint and `FALSE` after resuming the execution from the checkpoint.
- `copy` copies a file into another one.
- `diff` saves into the last file provided as a parameter the list of modifications that should be applied on the first file to obtain the second one.
- `merge` applies the list of modifications saved in the second file provided as a parameter to the checkpoint file provided as the first parameter.
- `wait_for` returns after the file which name is provided as a parameter is available.
- `restart` resumes the execution of the current process from the checkpoint file provided as a parameter.

Note that the operation that consists in resuming the execution of the first checkpoint generated in this example, the line in italic in Fig. 2, is executed on the master node but delegated to an external process in charge of managing the distribution of processes on a set of remote resources. BOINC [18], used in the scope of the Seti@Home project, is probably one of the most famous tool aiming at distributing works among a set of computing resources. For an in-depth description of CAPE, refer to [16] and [17].

V. New model of CAPE based on discontinuous incremental checkpoints

The performance analysis of the implementation of CAPE based on a complete checkpointer showed that the larger part of the program execution is spent in creating checkpoints, sending checkpoints over the network, computing the difference between two checkpoints, and injecting the previously computed difference into a process [19]. An optimization had been introduced by distributing the computation of the differences between two checkpoints on the set of nodes and then return to the master node the difference only instead of the complete checkpoint, but performance results still remained quite poor as at least one complete checkpoint had to be sent over the network. It clearly appeared that the unique viable solution consists in using incremental checkpoints only.

The main idea behind using incremental checkpoints is twofold: first, this allows to transmit far less data over the

network; second, the time needed to deal with incremental checkpoints is more interesting. For example, instead of creating a checkpoint and then compute the differences between this new checkpoint and another one that serves as a reference, it is now possible to directly generate the set of differences as it is the checkpoint itself. Moreover, the use of incremental checkpoints also allows to avoid the copy of complete checkpoints that was time consuming. Beside it, to avoid as much as possible the effect of checkpointer on the execution's speed of process [20], only some sections of program have to be checkpoited. This is the reason for the developping our discontinuous incremental checkpointing technique with the two additional functions are `start` and `stop`. In this technique, the checkpointing is not continuously executed from the begin of program but only in the scope of pairs `start` and `stop`. In each such as pair, many demands of creating incremental checkpoints can be made by using `create` function. However, we believe it might be possible to implement CAPE using incremental checkpoints on top of any checkpointing tool.

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

↓ automatically translated into ↓

```
if ( master ( ) )
      start ( )
      P1
      create ( delta1 )
      stop ( )
      wait_for ( delta2 )
      inject ( delta2 )
      if ( ! last_parallel ( ) )
            send ( delta1, slave )
else
      start ( )
      P2
      create ( delta2 )
      stop ( )
      send ( delta2, master )
      if ( ! last_parallel ( ) )
            receive ( delta1 )
            inject ( delta1 )
      else
            exit
```

Fig. 3. Template for OpenMP with discontinuous incremental checkpoints.

Figure 3 presents the new version of the piece of code that is substituted to an OpenMP parallel-sections construct. Beside the `start`, `stop` and `create` functions of checkpointer, the other new and changed functions are:

- `master` returns `TRUE` when executing on the master node and `FALSE` when executing on a slave.
- `last_parallel` returns `TRUE` is the current parallel block is the last one of the entire program and `FALSE` otherwise.
- `send` transfers the content of the file provided as the first parameter to the node provided as the second parameter.
- `receive` waits for the file provided as a parameter to be available.
- `inject` updates the current process with the information provided in the checkpoint file provided as a parameter. Note that this function does not update the instruction pointer.

In this prototype, the first change is the execution sequential sections of program on all nodes of systems. This change aim for avoiding the sending large size complete checkpoints to resum program on slave nodes. The resume of program on master node is also removed by using the `inject` function. Finally, the `deltas` now can be created directly by using the new `create` function, that comsumes less time and memory space than by using the complete checkpoint version of both `create` and `diff` functions.

It is important to note that the template presented in Fig. 3 can be applied several times the one after the other one inside a single program, or can be nested. Also note that the same mechanism may be applied to for loops to generate a distributed-memory version of the program.

VI. Performance evaluation

In order to validate our approach, some performance measurements have been conducted on a Desktop Grid. This testbed is composed of nodes including Intel(R) Core(TM)2 Duo E8400 CPUs running at 3 GHz and 2 GB RAM, and operated by Linux kernel 2.6.35 with Ubuntu 10.10 flavor and connected by a standard Ethernet network.

The program used for tests is a matrix-matrix product for which the size varies from 3,000×3,000 to 12,000×12,000. Matrices are supposed to be dense with integers and no specific algorithm has been implemented to take into account sparse matrices. Each experiment has been performed at least 10 times and a confidence interval of at least 90% has always been achieved for the measures. Data reported here are the means of the 10 measures.

Size	Sequential	OpenMP
3,000	258.9	142.4
6,000	1,852.7	1,048.7
9,000	7,314.5	3,986.2
12,000	14,990.5	8,999.4

TABLE I
Execution time (in seconds) on a single node.

The execution of both the sequential version and the OpenMP version of the program on one of the nodes gives the result provided in Table I. A single core was used for the sequential execution of the program, while the OpenMP program took benefits of the two cores. One can check that results in the Table I are consistent as the execution time for both sequential and OpenMP versions are directly proportional to the cube of the matrix size. Typically, this means that no important cache effects have polluted the performance measurements, probably because almost all data were fitting into memory. Moreover, the speedup obtained by OpenMP is 1.8 for the first three matrix sizes and 1.65 for the fourth one, which are expected values.

Figure 4 and 5 present the execution time in seconds of the matrix-matrix program for various number of nodes and matrix sizes. Note that, despite the fact that processors are dual core, a single core was used during the experiments. Three measures are represented each time: the left one is associated with CAPE using complete checkpoints, the middle one is also associated with CAPE but with incremental checkpoints, and the right one is associated with MPI. The MPI program has been developed for reference as exchanges to keep all processes consistent between nodes are kept minimal.

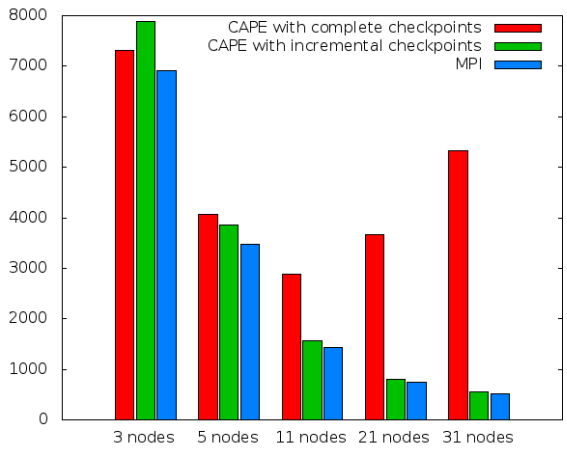

Fig. 4. Execution time (in seconds) vs. number of nodes.

Figure 4 presents the execution time for different number of nodes. The size of matrices are 12,000×12,000. However, similar trends are observed for the other matrix sizes. One can remark that the 3-node case apart, the execution time when using incremental checkpoints is always better than the execution time using complete checkpoints. The larger the number of nodes, the smaller the execution time for both CAPE using incremental checkpoints and MPI. Moreover, the execution time for CAPE using incremental checkpoints is getting closer and closer as the number of nodes is increasing. The case for CAPE using complete checkpoints is different. When few nodes are used for the computation (up to 11), the execution time is decreasing as the number of nodes is increasing and the value is quite similar to the other two cases (CAPE using incremental checkpoints and MPI). However, for larger number of nodes, the execution time for CAPE using complete checkpoints is directly proportional to the number of nodes. This is due to the amount of data that is transmitted over the network which is getting very important (there is at least one complete checkpoint for each slave node) even though the amount of data that are effectively interesting for each slave node is reduced. This clearly justifies the use of incremental checkpoints for CAPE.

At first, the performance for three nodes may look strange as the execution time of the program with CAPE using complete checkpoints is better than the execution time with CAPE using incremental checkpoints. In fact, for small number of nodes, the amount of data transmitted over the network between the different nodes is almost the same for both complete and incremental checkpoints as in the case of incremental checkpoints slave nodes receive a big part of matrices. However, in the case of incremental checkpoints, processes are monitored in order to capture the memory pages that are accessed for writing. The monitoring of the slave processes involves a computing overhead that is reduced proportionally with the amount of compution, and therefore with the number of nodes, when a large number of nodes is used. Fortunately, this is not a problem for CAPE. Processors with 4 and even 8 cores are available on the market and, as a result, CAPE is targeting architectures with a larger number of nodes.

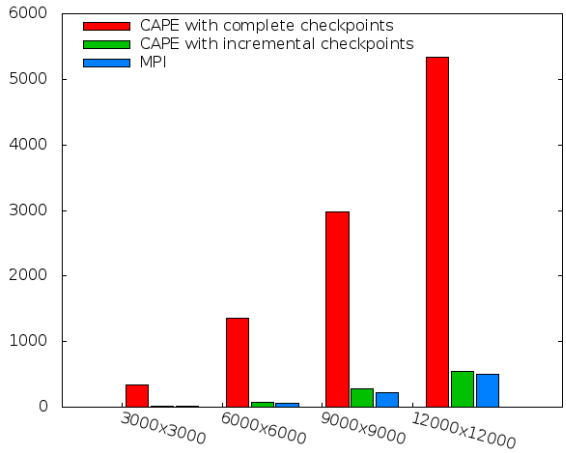

Fig. 5. Execution time (in seconds) vs. problem size.

Figure 5 presents the execution time for difference matrix sizes. The number of nodes involved in the parallel machine is 31. However, the remarks below would be the same with other number of nodes. The figure clearly shows that the execution time for CAPE using complete checkpoints is directly proportional to the square of the matrix size, while the execution time for both CAPE using incremental checkpoints and MPI is directly proportional to the matrix size. This is due to the fact that the virtual address space of the processes is mainly composed of the matrices, and that the complete virtual address space is transmitted over

the network for complete checkpoints. However, for CAPE using incremental checkpoints and MPI, the complete virtual address spaces are not transmitted over the network and only the data that have been updated during the computation of the matrix-matric product are considered. Moreover, one can remark that the execution time for CAPE using incremental checkpoints and MPI are very close. An in-depth analysis of the performance results shows that the execution time for CAPE using incremental checkpoints is only 10% higher than the execution time for MPI, except for 3,000×3,000 matrices where the ratio is 1.3 .

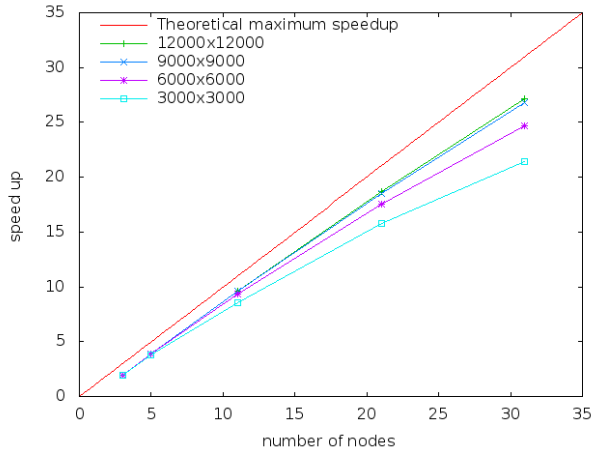

Fig. 6. Speedup vs. number of nodes.

Figure 6 shows the speedup of CAPE using incremental checkpoints for various number of nodes and matrix sizes. The red line represents the theoretical maximum speedup. The figure clearly shows that the solution is efficient with an efficiency (the ratio of the speedup over the number of nodes) in the range from 75% to 90%. Also, it highlights that the larger the size of the matrices, the higher the speedup, which was not the case with the complete checkpoint implementation.

VII. Conclusion and Future Works

This article presented CAPE and more specifically the modifications that have been applied on the template algorithm to translate automatically parallel programs with OpenMP directives into a parallel program targeted for distributed-memory architectures. Some performance analysis are also provided, proving the legitimity of the incremental checkpointing approach.

At present, CAPE has proven its efficiency for the generation of code satisfying the Bernstein's conditions for distributed-memory architecture. In the near future, we have planed to go further the Bernstein's conditions and take into account shared variables.

References

[1] John H. Merlin. *Distributed OpenMP: extensions to OpenMP for SMP clusters*. Proceedings of the Second European Workshop on OpenMP (EWOMP'00), Edinburgh, UK, September 2000.

[2] Sven Karlsson, Sung-Woo Lee, Mats Brorsson, Sahni Sartaj, Viktor K. Prasanna and Shukla Uday. *A fully compliant OpenMP implementation on software distributed shared memory*. Proceedings of the International Conference on High Performance Computing, Bangalore, India, LNCS 2552, pp. 195–206, December 2002.

[3] Andrzej M. Goscinski, Michael Hobbs and Jack Silcock. *GENESIS: an efficient, transparent and easy to use cluster operating system*. Journal of Parallel Computing, 28(4):557–606, April 2002.

[4] Roy Friedman, Maxim Goldin, Ayal Itzkovitz and Assaf Schuster. *MILLIPEDE: Easy parallel programming in available distributed environments*. Journal of Software Practice and Experience, 27(8):929–965, 1997.

[5] Eduardo Pinheiro and Ricardo Bianchini. *Nomad: A scalable operating system for clusters of uni and multiprocessors*. Proceedings of the 1st IEEE International Workshop on Cluster Computing, Melbourne, Australia, pp. 247–254, December 1999.

[6] Christine Morin, Renaud Lottiaux, Geoffroy Valle, Pascal Gallard, Gal Utard, R. Badrinath and Louis Rilling. *Kerrighed: A Single System Image Clustet Operating System for High Performance Computing*. Euro-Par 2003 Parallel Processing, Klagenfurt, Austria, LNCS 2790, pp. 1291–1294, August 2003.

[7] Christine Morin. *XtreemOS: a Grid Operating System Making your Computer Ready for Participating in Virtual Organizations*. Proceedings of the IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC), Santorini Island, Greece, Invited talk, May 2007.

[8] Contrail: Open Computing Infrastructure for Elastic Services. Web Site. `http://contrail-project.eu/`

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

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

[11] S. YI, J. Heo, Y. Cho, J. Hong, J. Choi and G. Jeon. *Ickpt: An Efficient Incremental Checkpointing Using Page Writing Fault - Focusing on the Implementation in Linux Kernel*. Proceedings of the ISCA 19th International Conference on Computers and Their Applications (CATA04), Seattle, WA, pp. 209-212, March 2004.

[12] James S. Plank, Micah Beck, Gerry Kingsley and Kai Li. *Libckpt: Transparent Checkpointing under Unix*. Proceedings of the Usenix Winter 1995 Technical Conference, New Orleans, LA, pp. 213–223, January 1995.

[13] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang and Fabrizio Petrini. *Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers*. Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, Seattle, WA, p. 9, November 2005.

[14] Jason Ansel, Kapil Arya, and Gene Cooperman. *DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop*. Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09), Rome, Italy, pp. 1–12, May 2009.

[15] Yuqun Chen, James S. Plank and Kai Li. *CLIP: a checkpointing tool for message-passing parallel programs*. Proceedings of the 1997 ACM/IEEE Conference on Supercomputing, New York, NY, p. 33, November 1997.

[16] Éric Renault. *Distributed Implementation of OpenMP Based on Checkpointing Aided Parallel Execution*. International Workshop on OpenMP (IWOMP), Beijing, China, LNCS 4935, pp. 183–193, June 2007.

[17] Éric Renault. *Parallelization of For Loops Using Checkpointing Techniques*. Proceedings of the 2005 International Conference on Parallel Processing Workshops, Oslo, Norway, pp. 313–319, June 2005.

[18] David P. Anderson. *BOINC: A System for Public-Resource Computing and Storage*. Proceedings of 5th IEEE/ACM International Workshop on Grid Computing, Pittsburg, PA, pp. 4–10, November 2004.

[19] Laura Mereuta and Éric Renault. *Checkpointing Aided Parallel Execution Model and Analysis*. High Performance Computation Conference (HPCC), Houston, TX, LNCS 4782, pp. 707–717, September 2007.

[20] Viet Hai Ha and Éric Renault. *Discontinuous Incremental: A New Approach Towards Extremely Lightweight Checkpoints*. Proceedings of the International Symposium on Computer Networks and Distributed Systems (CNDS), Tehran, Iran, February 2011.