Impact of Over-Decomposition on Coordinated Checkpoint/Rollback Protocol

Xavier Besseron[1] and Thierry Gautier[2]

[1] Dept. of Computer Science and Engineering, The Ohio State University
besseron@cse.ohio-state.edu
[2] MOAIS Project, INRIA
thierry.gautier@inrialpes.fr

Abstract. Failure free execution will become rare in the future exascale computers. Thus, fault tolerance is now an active field of research. In this paper, we study the impact of decomposing an application in much more parallelism that the physical parallelism on the rollback step of fault tolerant coordinated protocols. This over-decomposition gives the runtime a better opportunity to balance workload after failure without the need of spare nodes, while preserving performance. We show that the overhead on normal execution remains low for relevant factor of over-decomposition. With over-decomposition, restart execution on the remaining nodes after failures shows very good performance compared to classic decomposition approach: our experiments show that the execution time after restart can be reduced by 42 %. We also consider a partial restart protocol to reduce the amount of lost work in case of failure by tracking the task dependencies inside processes. In some cases and thanks to over-decomposition, this partial restart time can represent only 54 % of the global restart time.

Keywords: parallel computing, checkpoint/rollback, over-decomposition, global restart, partial restart.

1 Introduction

The number of components involved during the execution of High Performance applications keeps growing. Thus, the probability of failure during an execution is very high. Fault tolerance is now a well studied subject and many protocols has been provided [8]. The coordinated checkpoint/rollback scheme is widely used, principally because of its simplicity, in particular in [5,10,13,23]. However, among other drawbacks, the coordinated checkpoint/rollback approach suffers from the following issues.

1. Lack of flexibility after restart. Coordinated checkpoint implies that the application will be recovered in the same configuration. Three approaches exist to restart the failed processes. The first one is to wait for free nodes or for

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part II, LNCS 7156, pp. 322–332, 2012.

failed nodes to be repaired. To avoid waste of time, the second approach is based on spare nodes which are pre-allocated before the beginning of execution. Third, the application is restarted only on the remaining nodes (over-subscription). However, without redistribution of the application workload, this approach leads to poor performance of the execution after restart.
2. Lost work because of the restart. The global restart technique associated to the coordinated checkpoint requires all the processes to rollback to their last checkpoint in case of failure, even those which did not failed. Then, a large amount of computation has to be executed twice, which constitutes a waste of time and computing resources.

Addressing these issues is a challenging task. To overcome these limitations, our proposition is based on the over-decomposition technique [19,4,18], coupled with a finer representation of the internal computation with a data flow graph. Thanks to this, the scheduler can better balance the workload. The contributions of this work are: 1. We leverage over-decomposition to restart an application on a smaller number of nodes (*i.e.* without spare node) while preserving well-balanced workload in order to lead to a better execution speed. 2. We combine over-decomposition and the partial restart technique proposed in [2] to reduce the time required to re-execute the lost work. Thus, it speeds up the recovery time of the application after a failure. 3. We propose an experimental evaluation of these techniques using the Kaapi [11] middleware.

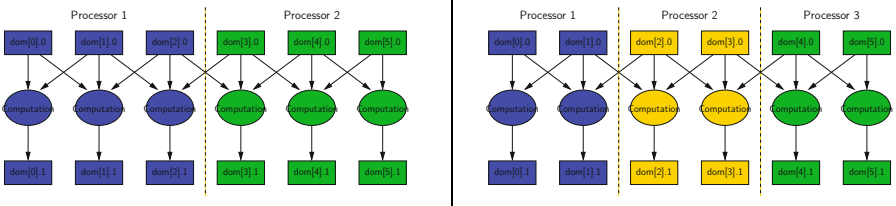
This paper is organized as follow. The next Section gives background about the over-decomposition principle, the Kaapi data flow model and coordinated checkpoint/restart. The Section 3 explains how over-decomposition can benefit to the global and partial restart techniques. Experiments and evaluations are presented in Section 4. Then, we give our conclusions.

2 Background

This work was motivated by the parallel domain decomposition applications. In the remaining of the paper, we consider an iterative application called Poisson3D which solves the Poisson's partial differential equation with a 7-point stencil over a 3D domain using finite difference method. The simulation domain is decomposed in d sub-domains. Then, the sub-domains are assigned to processes for the computation (classically one sub-domain per MPI process).

Kaapi and Data Flow Model. Kaapi[1] [11] is a task-based model for parallel computing inherited from Athapascan [9]. Using the access mode specifications (read, write) of the function-task parameters, the runtime is able to dynamically compute the data flow dependencies between tasks from the sequence of function-task calls, see [9,11]. These dependencies are used to execute concurrently independent tasks on the idle resources using work stealing scheduling [11]. Furthermore, this data flow graph is used to capture the application state for many original checkpoint/rollback protocols [14,2].

[1] http://kaapi.gforge.inria.fr



(a) Scheduling on 2 processors (b) Scheduling on 3 processors

Fig. 1. Example of over-decomposition: the same data flow graph, generated for 6 sub-domains, is scheduled on 2 or 3 processors

In the context of this paper, Kaapi schedules an over-decomposed data flow graph using a *static scheduling* strategy [15,11]. Once computed for a loop body, the data flow graph scheduling can be re-used through several iterations until the graph or the computing resources change.

Over-decomposition. The over-decomposition principle [19] is to choose a number d of sub-domains significantly greater than the number n of processors [19,18], *i.e.* $d \gg n$. Once d fixed, it defines the parallelism degree of the application. Thanks to this, the scheduler has more freedom to balance the workload among the processors. Figure 1 shows a basic example of how a simple over-decomposed data flow graph can be partitioned by Kaapi using static scheduling.

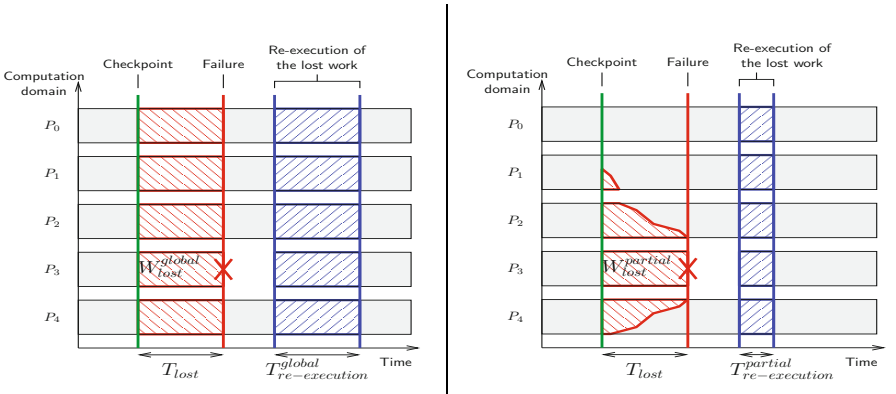
Coordinated Checkpoint/Rollback. The coordinated checkpoint/rollback technique [8,22] is composed of two phases. During the failure-free execution, the application is periodically saved. The processes are coordinated to build a consistent global state [7], and then they are checkpointed. In case of failure, the application is recovered using the last valid set of checkpoints. This last step requires all the processes to rollback. It is called a global restart.

3 Over-Decomposition for Rollback

Kaapi provides a fault-tolerance protocol called CCK for Coordinated Checkpoint in Kaapi [2]. It is based on the blocking coordinated checkpoint/rollback protocol, originally designed by Tamir et al. in [22]. CCK provides two kinds of recovery protocol. The *global restart* is the classic recovery protocol associated with the coordinated checkpoint approach. The *partial restart* is an original recovery presented in [2] which takes advantage of the data flow model.

3.1 Over-Decomposition for Global Restart

The global restart protocol of CCK works like the recovery of the classic coordinated checkpoint/rollback protocol: once a failure happens, all the processes rollback to their last checkpoint. During this step, and contrary to most of the other works, we do not assume that spare nodes are available to replace the failed nodes. In a such case, standard MPI programming model would impose



(a) Global restart (b) Partial restart

Fig. 2. Lost work and time to re-execute the lost work for global restart and partial restart

to restart many processes on the same core (over-subscription), and that would lead to poor execution performance after restart.

With Kaapi, an application checkpoint is made of its data flow graph [11,14]. Then it is possible to balance the workload after restart on the remaining processes, without requiring new processes or new nodes. The over-decomposition allows the scheduler to freely re-map tasks and data among processors in order to keep a well-balanced workload. Experimental results on actual executions of the Poisson3D application are presented in Section 4.1.

3.2 Over-Decomposition for Partial Restart

The partial restart for CCK [2] assumes that the application is checkpointed periodically using a coordinated checkpoint. However, instead of restarting all the processes from their last checkpoint as for global restart, partial restart only needs to re-execute a subset of the work executed since the last checkpoint to recover the application. We call the *lost work*, the computation that has been executed before the failure, but that needs to be re-executed in order to restart the application properly. W_{lost}^{global} is the lost work for global restart on Figure 2a and $W_{lost}^{partial}$ represents the lost work for partial restart on Figure 2b.

To allow the execution to resume properly, and similarly to the message logging protocols [8], the non-failed processes have to replay the messages that have been sent to the failed processes since their last checkpoint. Since these messages have not been logged during execution, they will be regenerated by re-executing a subset of tasks on the non-failed processes. This *strictly required* set of tasks is extracted from the last checkpoint by tracking the dependencies inside the data flow graph [2].

This technique is possible because the result of the execution is directed by a data flow graph where the reception order of the message does not impact the computed value [9,11]. As a result, this ensures that the restarted processes will reach exactly the same state as the failed processes before the failure [2].

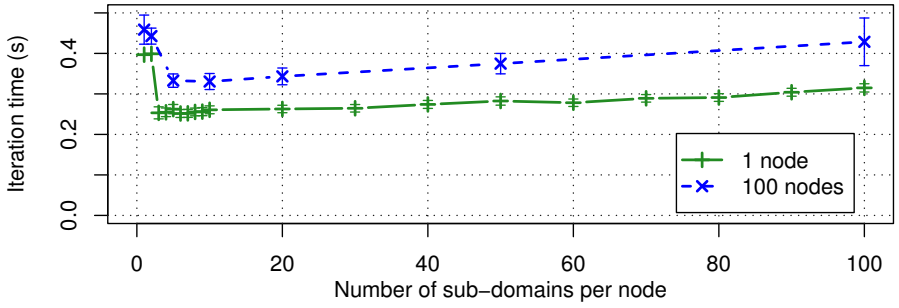


Fig. 3. Iteration time in function of the number of sub domains per node with a constant domain size per node (lower is better)

With over-decomposition, the computation of each process is composed by a large amount of small computation tasks in our data flow model. This gives freedom to the scheduler to re-execute the lost work in parallel. The size of the lost work has been studied theoretically and using simulations in [2]. The experiments of the Section 4.2 show results on actual executions.

4 Experimental Results

We evaluate experimentally the techniques proposed in this paper with the Poisson3D application sketched in Section 2. The amount of the computation in each iteration is constant, so the iteration time remains approximately constant between steps. The following experiments report the average iteration time.

Our experimental testbed is composed on the Griffon and Grelon clusters located at Nancy and part of the Grid'5000 platform[2]. The Griffon cluster has 92 nodes with 16 GB of memory and two 4-core Intel Xeon L5420. The Grelon cluster is composed of 120 nodes with 2 GB of memory and two 4-cores Intel Xeon 5110. All nodes from the both clusters are connected with a Gigabit Ethernet network and 2 level of switches.

Over-decomposition overhead. Over-decomposition may introduce an overhead at runtime due to the management of the parallelism. The purpose of this first experiment is to measure this overhead. We use a constant domain size per core: 10^7 double-type reals, *i.e.* 76 MB and we vary the decomposition d, which is the number of sub-domains per core. We run this on 1 and 100 nodes. In both cases, we use only 1 core for computation on each node to simplify the result analysis.

Figure 3 shows the results of the experiment on the Grelon cluster. Each point is the average value of one hundred iterations and the error-bars show the standard deviation. With one node, the iteration time for decomposition in 1 or 2 sub-domains is about 0.4 s. For 3 sub-domains, the execution time drops by 35 % due to better cache use with small blocks. For higher decomposition factors, the iteration time slowly increases linearly. It is the overhead due to the

[2] http://www.grid5000.fr

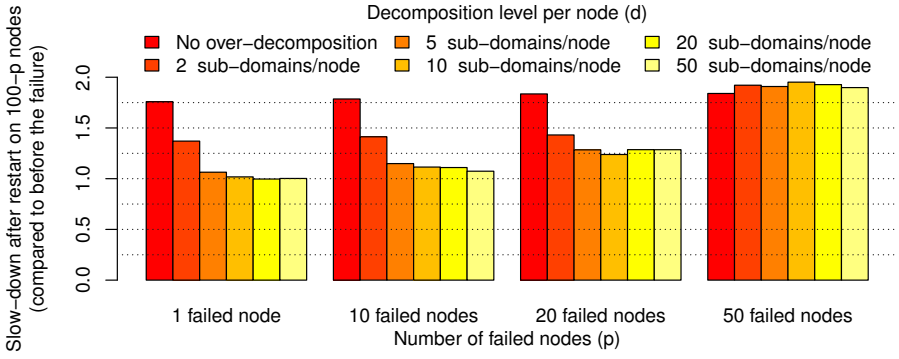


Fig. 4. Slow-down after restart on $100 - p$ nodes compared to the execution before the failures for different decompositions d (lower is better)

management of this higher level of parallelism. Compared to the best value (*i.e.* for 3 sub-domains per node), the overhead is around 3 % for 10 sub-domains per node and 25 % for 100 sub-domains. The curve shape with 100 nodes is similar but it is shifted up, between 0.05 and 0.1 seconds higher, due to the communication overhead.

4.1 Global Restart

We measure the gain on the iteration time due to the capacity to reschedule the workload after the failure of p processors. We consider the following scenario: The application is executed on n nodes with periodic coordinated checkpoint. Then, p nodes fail[3]. The application is restarted on $n - p$ nodes using the global restart approach and the load-balancing algorithm is applied.

Execution speed after restart. We run the Poisson3D application on $n = 100$ nodes of the Grelon cluster (using only 1 core per node) with coordinated checkpoint and we use a total domain size of 10^9 doubles, *i.e.* 7.5 GB. Then, after the failure of p nodes, the application is restarted on $100 - p$ nodes and the sub-domains are balanced among all the remaining nodes. We measured the iteration time of the application before and after the failure and then got the average of 100 values. Figure 4 reports the slow-down (iteration time after failure over iteration time before failure) for different decomposition factors d.

First, we have to notice that the execution after restart is always slowed-down, because it uses a smaller number of nodes. For example, after the failure of 50 nodes, the execution is almost 2 times slower because it now uses 50 nodes instead of 100 nodes. The lost of half of the nodes slows down by a factor less than 2 the execution after failure due to fewer messages communicated on less nodes.

When using 10, or more, sub-domains per node, the slow-down is considerably reduced, in particular when the number of failed nodes is not a divisor of the

[3] A node failure is simulated by killing all the processes on the node.

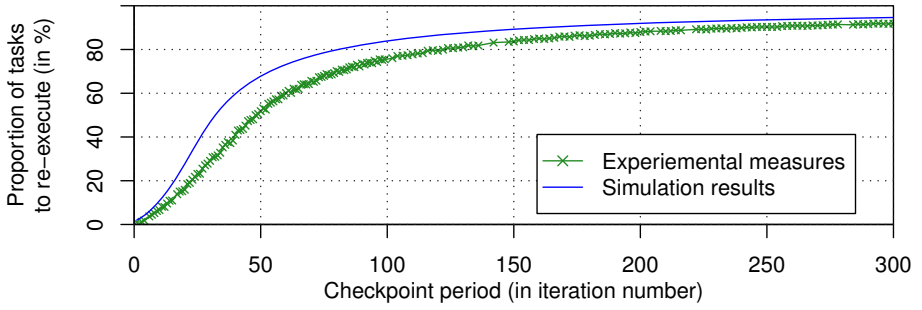


Fig. 5. Proportion of lost work for the partial restart in comparison to the classic global restart approach (lower is better)

initial node number. When only one node failed, our results shows that execution time after restart with over-decomposition is reduced by 42 %. We want to emphasize that this improvement applies to all the iterations after the restart. Thus, it will be beneficial to all the rest of the execution.

Load-balancing cost. This experiment evaluates the cost of the restart and load-balancing steps. We measure this cost on the Griffon cluster using 80 nodes with 8 cores, *i.e.* 640 cores, each with a domain size of 10^6 doubles per core, *i.e.* 7.6 MB. One node fails and the application is restarted on 79 nodes. The $7s$ of the global restart time is decomposed as following: $2.1s$ for the process coordination time and the checkpoint loading time; $1.7s$ to compute and apply the new schedule; and finally, data redistribution between processes takes $3.2s$.

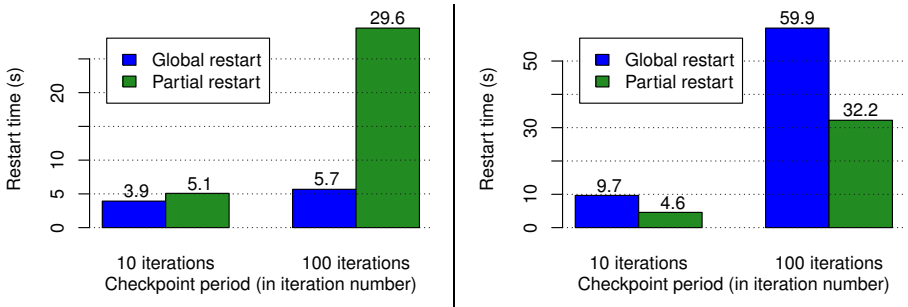
4.2 Partial Restart

In this Section, we focus on partial restart. The application runs on 100 nodes and one node fails. It is then restarted on 100 nodes using one spare node.

Lost work. Figure 5 reports the proportion of tasks to re-execute with respect to the global restart, *i.e.* $W_{lost}^{partila} / W_{lost}^{global}$. These values correspond to the worst case, *i.e.* when the failure happens just before the next checkpoint.

The X-axis represents the checkpoint period (in iteration count). A smaller period implies frequent checkpoints and few dependencies between sub-domains and thus a smaller number of tasks to re-execute. A bigger period lets the dependencies between sub-domains be propagated among the sub-domains and processors (for domain decomposition application, the communication pattern introduces local communications): more tasks need to be re-executed to recover. Experimental measures ran on 100 nodes of the Grelon cluster with 16 sub-domains per node. Simulation results come from [2].

Partial restart time. The results of the Figure 6 shows the restart time of global and partial restart. For checkpoint periods of 10 or 100 iterations, the lost work of partial restart represents respectively 6 % and 51 % of the lost work of the global restart. We have used 100 nodes of Grelon and a total domain size of 76 MB and we restarted the application on the same number of nodes. Using



(a) Sub-domain computation ≈ 2 ms (b) Sub-domain computation ≈ 50 ms

Fig. 6. Comparison of the restart time between global restart and partial restart, for different checkpoint periods and different computation grains (lower is better)

two different sub-domain computations allows to see the influence of the data redistribution (the data size and the communication volume are kept identical).

The restart time for partial restart include the time to re-execute the lost work, *i.e.* the strictly required set of tasks, and also the time to redistribute the data, which can be costly. It is difficult to measure these two steps independently because they are overlapped in the Kaapi implementation. For the global restart, this time corresponds only to the time to re-execute the lost work: in this experiment, there is no need to redistribute the data because the workload remains the same as before the failure.

For a small computation grain, *i.e.* a sub-domain computation time of 2 ms, the performance of partial restart is worse than global restart because the data redistribution represents most of the cost of the partial restart, mainly because the load-balancing algorithm used does not take in account the data locality. For a coarser grain, *i.e.* a sub-domain computation time of 50 ms, the partial restart achieves better performance. For a 100-iteration checkpoint period, the partial restart time represents only 54 % of the global restart time (for a lost work which corresponds to 51 %).

5 Related Works

The over-decomposition principle is to decompose the work of an application in a number of tasks significantly greater than the number of computing resources [19,18]. Over-decomposition is applied to many programming models in order to hide communication latency by computation (Charm++ [18]), to simplify scheduling of independent tasks (Cilk [3]) or with data flow dependencies (PLASMA [21] and SMPSS [1] which do not consider recursive computation; or Kaapi [9,11] that allows recursive data flow computation). For the MPI programming model, the parallelism description is tightly linked to the processor number. Alternatives are AMPI [18], or hybrid approaches like MPI/OpenMP [20] or MPI/UPC [16] which make use of over-decomposition.

On the fault tolerance aspect, most of the works focus in the message passing model. Many protocols, like checkpoint/rollback protocols and message logging protocols, has been designed [8] and they are widely used [5,10,13,23,6]. In [12], communication determinism is leveraged to propose optimized approaches for certain application classes.

Charm++ can use over-decomposition to restart an application only on the remaining nodes after a failure [17] with coordinated checkpoint/restart and message logging approaches. It relies on a dynamic load-balancing algorithm which periodically collects load information of all the nodes and redistributes the Charm++ objects if required.

Similarly to Charm++, our work in Kaapi allows to restart an application on the remaining nodes using over-decomposition. Additionally, we leverage over-decomposition to reduce the restart time of the application thanks to the original partial restart approach. Also in our work, we consider a data flow model which allows a finer representation of the application state. Furthermore, our load-balancing algorithm is based on the data flow graph of the application and is executed only after the restart.

6 Conclusion and Future Work

We presented the impact of over-decomposition on execution after failure using the classical checkpoint/rollback scheme. First, when an application is restarted without spare node, over-decomposition allows to balance the workload. In our experimental results, the execution time after restart with over-decomposition is reduced by 42 % compared to a decomposition based on the process number. Furthermore, the improvement benefits to all the rest of the execution.

Secondly, we leverage over-decomposition to improve the restart with the partial restart technique proposed in [2]. The partial restart allows to reduce the amount of work required to restart an application after a failure. The over-decomposition exposes the parallelism residing in this lost work. In our experimental results, we showed that, in some cases, the partial restart time represents only 54 % of the global restart time. The experiments also highlight that the data redistribution induced by the load-balancing can have a significant impact on the partial restart performance.

For future work, it is planned to extend the partial restart algorithm to support the uncoordinated checkpoint approach. Thus, this will avoid the I/O contention due to the coordinated checkpoint. Additionally, load-balancing algorithms that take in account data movement will be studied to improve the partial restart.

Acknowledgment. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

1. Badia, R.M., Herrero, J.R., Labarta, J., Pérez, J.M., Quintana-Ortí, E.S., Quintana-Ortí, G.: Parallelizing dense and banded linear algebra libraries using smpss. Concurr. Comput. : Pract. Exper. (2009)
2. Besseron, X., Gautier, T.: Optimised recovery with a coordinated checkpoint/rollback protocol for domain decomposition applications. In: MCO 2008 (2008)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Parallel and Distributed Computing (1996)
4. Bongo, L.A., Vinter, B., Anshus, O.J., Larsen, T., Bjorndalen, J.M.: Using overdecomposition to overlap communication latencies with computation and take advantage of smt processors. In: ICPP Workshops (2006)
5. Bouteiller, A., Hérault, T., Krawezik, G., Lemarinier, P., Cappello, F.: MPICH-V project: a multiprotocol automatic fault tolerant MPI. High Performance Computing Applications (2006)
6. Chakravorty, S., Kale, L.V.: A fault tolerant protocol for massively parallel systems. In: IPDPS (2004)
7. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems (1985)
8. Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys (2002)
9. Galilée, F., Roch, J.L., Cavalheiro, G., Doreille, M.: Athapascan-1: On-line building data flow graph in a parallel language. In: PACT 1998 (1998)
10. Gao, Q., Yu, W., Huang, W., Panda, D.K.: Application-transparent checkpoint/restart for mpi programs over infiniband. In: ICPP 2006 (2006)
11. Gautier, T., Besseron, X., Pigeon, L.: Kaapi: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO 2007 (2007)
12. Guermouche, A., Ropars, T., Brunet, E., Snir, M., Cappello, F.: Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. In: IPDPS (2011)
13. Hursey, J., Squyres, J.M., Mattox, T.I., Lumsdaine, A.: The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In: IPDPS (2007)
14. Jafar, S., Krings, A.W., Gautier, T.: Flexible rollback recovery in dynamic heterogeneous grid computing. IEEE Transactions on Dependable and Secure Computing (2008)
15. Jafar, S., Pigeon, L., Gautier, T., Roch, J.L.: Self-adaptation of parallel applications in heterogeneous and dynamic architectures. In: ICTTA 2006 (2006)
16. Jose, J., Luo, M., Sur, S., Panda, D.K.: Unifying UPC and MPI Runtimes: Experience with MVAPICH. In: PGAS 2010 (2010)
17. Kale, L.V., Mendes, C., Meneses, E.: Adaptive runtime support for fault tolerance. Talk at Los Alamos Computer Science Symposium 2009 (2009)
18. Kale, L.V., Zheng, G.: Charm++ and AMPI: Adaptive runtime strategies via migratable objects. In: Advanced Computational Infrastructures for Parallel and Distributed Applications. Wiley-Interscience (2009)
19. Naik, V.K., Setia, S.K., Squillante, M.S.: Processor allocation in multiprogrammed distributed-memory parallel computer systems. Parallel Distributed Computing (1997)

20. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: PDP 2009 (2009)
21. Song, F., YarKhan, A., Dongarra, J.: Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: SC 2009 (2009)
22. Tamir, Y., Séquin, C.H.: Error recovery in multicomputers using global checkpoints. In: ICPP 1984 (1984)
23. Zheng, G., Shi, L., Kale, L.V.: FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. Cluster Computing (2004)