Persistent Fault-Tolerance for Divide-and-Conquer Applications on the Grid

Gosia Wrzesinska, Ana-Maria Oprescu, Thilo Kielmann, and Henri Bal

Vrije Universiteit Amsterdam
{gosia,amo,kielmann,bal}@cs.vu.nl

Abstract. Grid applications need to be fault tolerant, malleable, and migratable. In previous work, we have presented *orphan saving*, an efficient mechanism addressing these issues for divide-and-conquer applications. In this paper, we present a mechanism for writing partial results to checkpoint files, adding the capability to also tolerate the *total loss* of all processors, and to allow suspending and later resuming an application.

Both mechanisms have only negligible overheads in the absence of faults, even with extremely short checkpointing intervals like one minute. In the case of faults, the new checkpointing mechanism outperforms orphan saving by 10 % to 15 %. Also, suspending/resuming an application has only little overhead, making our approach very attractive for writing grid applications.

1 Introduction

In grid environments, the availability of computing resources changes constantly. Processor crashes are more likely to occur than in traditional parallel environments. Also, processors may be taken away from an application because they are claimed by another, higher-priority application, or because a processor reservation has ended. At the same time, new processors may become available.

A grid application must be able to adapt to such changes in order to survive in a grid environment and to achieve good performance. In particular, three issues have to be dealt with: *Fault tolerance,* which is the ability of an application to operate in the presence of hardware and software failures. *Malleability,* which is the ability of an application to handle processors joining and leaving an on-going computation. And *migratability,* which is the ability of an application to relocate to a different set of computational resources during the run.

These three issues are closely related to each other. For example, if an application can handle crashing processors (fault tolerance) and continue working on the diminished number of processors, it can also handle leaving processors (partial malleability). Even more, if the processors are leaving *gracefully* (i.e., after a prior notice) handling it may be more efficient than handling crashing processors. Further, if an application is malleable, it is also migratable: it can be migrated from one set of resources to another by first adding the new processors to the computation and then removing the old ones.

A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.): Euro-Par 2007, LNCS 4641, pp. 415–426, 2007.

In previous work, we have presented an efficient mechanism supporting fault-tolerance, malleability, and migration of divide-and-conquer applications [1], in the following called *orphan saving*. It is based on re-executing jobs done by processors that have either crashed or left voluntarily, while preserving as many partial results as possible in the application's main memory. With this mechanism, applications are guaranteed to complete successfully, as long as at least one processor is alive, at any moment during the execution.

In this paper, we complement orphan saving by a mechanism for writing partial results to checkpoint files, adding the capability to also tolerate the *total loss* of all processors, and to allow suspending an application and to resume it later, whenever CPU's become available again.

Our performance evaluation shows that both fault-tolerance mechanisms have negligible overhead in the absence of faults. Suspending and resuming the tested applications has only 1 % – 11 % overhead, compared to uninterrupted runs. In the case of faults, the new checkpointing mechanism even outperforms orphan saving.

In Section 2, we briefly introduce the divide-and-conquer model, along with the *orphan saving* fault-tolerance mechanism. Section 3 presents our new, checkpointing fault-tolerance mechanism. In Section 4, we evaluate both mechanisms using two application programs. Related work is discussed in Section 5. In Section 6, we draw our conclusions.

2 Divide-and-Conquer and the Orphan Saving Mechanism

Divide-and-conquer applications operate by recursively dividing a problem into subproblems. The recursive subdivision goes on until the subproblems become trivial to solve. After solving subproblems, their results are recursively combined until the final solution is assembled. This leads to an *execution tree* of nested tasks. The excellent suitability of the divide-and-conquer paradigm for writing grid applications has been shown many times before [2,3,4,1].

We have implemented our fault-tolerance mechanisms within Satin, a Java framework for creating grid-enabled divide-and-conquer applications. With Satin, the programmer annotates the sequential code with divide-and-conquer primitives (marker interfaces and a synchronization method). Satin's byte-code rewriter generates the necessary communication and load-balancing code. In the following, we are using Satin to discuss and evaluate our fault-tolerance mechanisms.

In Satin, invocations of annotated divide-and-conquer methods lead to the creation of entries in the processor's local *work queue*. Work is distributed across the processors by work stealing: when a processor runs out of work, it picks another processor at random and steals a job from its work queue. After computing the job, the result is returned to the originating processor. Satin uses a very efficient, grid-aware load balancing algorithm which hides wide-area latencies by overlapping local and remote stealing [3]. Though the combination of local and remote

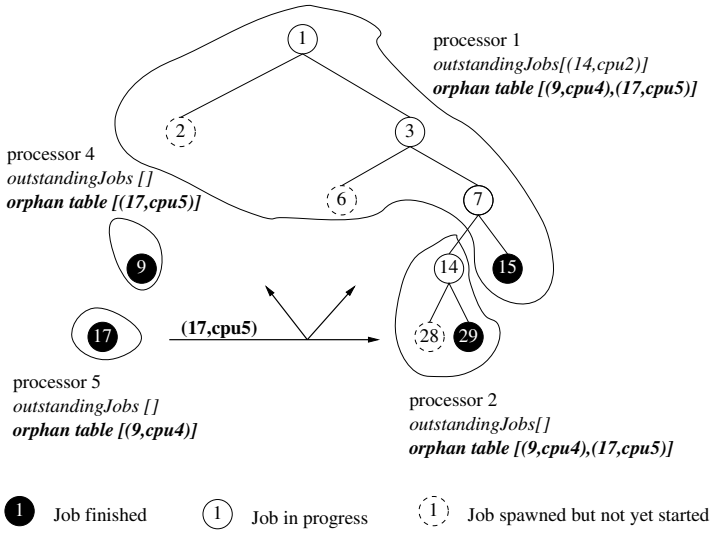

Fig. 1. *Orphan saving* re-using orphan results

stealing could allow for a job to be stolen *from* two or more processors in turn, the problem of livelock does not arise, simply because the job will eventually be stolen by a processor with an empty *work queue*, where the job will get executed immediately.

Adding a new machine to a divide-and-conquer computation is straightforward: the new machine simply starts stealing jobs from other machines. Removing processors (both voluntarily and in case of faults) can be handled by recomputing the work stolen by leaving processors. Implementing such a recomputation scheme efficiently, however, is not trivial, due to the problem of *orphan jobs*: those jobs that have been stolen *from* a leaving processor. The novelty of our *orphan saving* mechanism [1] is its ability to efficiently re-use the results of orphan jobs in case of a fault. This is achieved by carefully inserting the results of orphan jobs into the execution tree once the lost jobs have been resubmitted.

An example is sketched in Figure 1. Here, processor 3 has crashed. In response, processor 1 has re-submitted jobs 2 and 6 (once stolen *by* processor 3) to its work queue, while processors 4 and 5 have notified all processors that they have results for jobs 9 and 17 (once stolen *from* processor 3) available. As it is not known beforehand which processor will need an orphan result, all processors have to be notified where it could be found. As soon as an orphan job is up for recomputation, the processor in charge will check its orphan table first, thus re-using the orphan results.

Divide-and-conquer is extremely robust because utilizing orphan results is merely a performance optimization, albeit an important one. In case of several errors, leading to the loss of some or all orphan results, the divide-and-conquer application will still compute the correct result, only its performance will be impacted, depending on the amount of losses.

Orphan saving is exclusively using the state inside the main memory of the surviving (not leaving) processors to recover from faults. This mechanism is sufficient to guarantee the successful completion of an application, as long as at least one processor keeps alive. The special case of a crash of the processor with the root job, below called the *master*, needs a separate mechanism; in this case, Satin uses a traditional coordinator re-election mechanism.

In [1], we have shown that orphan saving works very efficiently, both in the absence and in the presence of failures. The mechanism can handle fault-tolerance, malleability, and migration. Its limitation is that it depends on at least one processor at a time being alive, from application startup to successful completion. This excludes both the case of "total loss" of all processors and the ability to suspend and later resume an application. In the following, we present a complementary scheme, using checkpointing to persistent storage, that overcomes these limitations.

3 Checkpointing Divide-and-Conquer Applications

To overcome the limitation of storing all orphan results in volatile main memory, we have developed a mechanism that stores intermediate results, just like orphan results, but on stable storage. Using the orphan result data structure allows us to build a very light-weight checkpointing scheme, comparable to application-level checkpointing, except that the Satin runtime system is taking care of storing the relevant data. This combines the advantages of system-level checkpointing (user transparency) and of application-level checkpointing (efficiency due to low data volume.)

All processors of a divide-and-conquer application periodically save their partial results in a *checkpoint file*. Along with a job's result, both *jobID* and originating *processorID* are stored. Each processor writes its own checkpoint asynchronously from the others, avoiding synchronization overheads. This is possible due to the robustness of divide-and-conquer and orphan saving, as explained in Section 2. The checkpointing interval is user defined.

Processors do not access the checkpoint file directly. Instead, they send their data to a centralized *coordinator* processor that is in charge of reading and writing the actual file. The coordinator's role is twofold:

1. **Fault tolerance:** If a processor crashes, the coordinator searches the checkpoint file for results computed by that processor. Those results are fetched into the coordinator's memory. Next, the orphan saving mechanism is used to re-use those results; they are treated just like orphan jobs. For each of those results, the coordinator forwards the *jobID* along with *its own processorID* to the other processors, allowing them to integrate these results into the job tree, as necessary.
2. **Suspend/resume:** When a computation is started, the coordinator checks whether the user-specified checkpoint file already exists. If so, the coordinator assumes that the computation has been restarted. All results from the

checkpoint file are read into the memory of the coordinator and for each of them, the *jobID* and the coordinator's *processorID* are sent to the other processors.

3.1 The Checkpoint File

The checkpoint file is accessed by the coordinator, but it need not necessarily be located on the coordinator's local filesystem. In fact, the user may specify an arbitrary location for the checkpoint file. Access to the checkpoint file is implemented using the Java implementation of the Grid Application Toolkit (GAT) [5]. The GAT provides transparent access to various grid middleware systems. For accessing remote files, a programmer only needs to specify a URI referring to the location. The GAT then takes care of selecting and using the appropriate protocol, like, for example, FTP, SSH, HTTP, or GridFTP.

The results stored in a checkpoint file are partially redundant. This is caused by the fact that many jobs are stored in the checkpoint file, along with their direct parent or another ancestor job. In such cases, only the ancestor is useful. Depending on the checkpoint interval, there may be more or less redundancy. We use *checkpoint compression* to reduce the number of such redundant jobs in a checkpoint file.

3.2 The Coordinator

Initially, the master is elected as checkpointing coordinator. To achieve good I/O performance, however, the coordinator is re-elected from among the processors taking part in the computation, based on actual I/O performance with the checkpoint file. For this purpose, each processor measures the time it takes to write a small file to the location of the checkpoint file. The master collects these results and selects the processor with the shortest file write time as the new coordinator.

If the coordinator crashes, a new coordinator has to be elected. The new election is initiated by the master, which sends a *coordinator reelection* message to all processors. Then, the above coordinator election procedure is performed. The processors postpone checkpointing until the election is completed.

If the coordinator has crashed while another process was sending checkpoint data to it, this data will be lost. Because loss of checkpoint data may only affect performance of the application but never its correctness, we do not take any action to avoid such situations.

The coordinator may also crash while writing to the checkpoint file and the checkpoint file may be corrupted. Therefore, each time a coordinator is initialized, it inspects the checkpoint file for possible errors. If errors are found, a new checkpoint file containing all non-damaged results is created and used.

We are using two different mechanisms to detect coordinator crashes: one implemented in the communication layer (Ibis), another one implemented in the Ibis Registry. If one of both mechanisms would lead to false positives, still this would only affect the *performance* of our system, but not its *correctness*.

To minimize the overhead of checkpointing, we use *concurrent checkpointing* [6]. The results are written to the checkpoint file by a separate thread in the coordinator process. This thread runs concurrently with the Satin computation.

There is no synchronization between the workers and the coordinator in terms of sending/receiving the checkpoint data.

4 Evaluation

We will now evaluate the performance of our fault-tolerance mechanisms, both in the absence and in the presence of faults. We compare the orphan-saving algorithm to the new checkpointing mechanism, the latter with various checkpointing intervals.

The experiments were carried out on the Distributed ASCI Supercomputer (DAS-2), consisting of five clusters that are located at universities in the Netherlands. We have used a total of 32 nodes on two DAS-2 clusters (16 nodes each). Each node contains two 1-GHz Pentium-III's and at least 1 GB RAM. All nodes run RedHat Linux. Within a cluster, nodes are connected by 2 Gb/s Myrinet. For intra-cluster communication we have used 100 Mb/s Ethernet and SurfNet, the Dutch academic Internet backbone. The bandwidth between the clusters is about 700 Mb/s. The round-trip latencies are around 2 ms.

In order to gain broad insights about our checkpointing mechanism, we have selected one computation-intensive application (TSP) and one communication-intensive code (Raytracer). The code for the Traveling Salesman Problem (*TSP*) searches for a shortest path connecting a set of cities. TSP is a well-known, NP-complete problem that has many applications in science and engineering. TSP was parallelized by evaluating different paths in parallel. TSP is a computation-intensive application and sends only little data. *Raytracer* renders a bitmap image from an abstract scene description. Raytracer has been parallelized by recursively subdividing the bitmap into smaller parts, and rendering the parts in parallel. Raytracer is a relatively communication-intensive application.

4.1 Performance Overhead in the Absence of Crashes

First, we assess the overhead on application performance, caused by either orphan saving or checkpointing, when no processors are leaving or crashing. For both applications, we compare the plain Satin system (without any fault tolerance) to Satin with orphan saving, and to Satin with checkpointing, using checkpointing intervals of one, two, and five minutes. Note that these checkpointing intervals are rather short, compared to traditionally used values, like 30 minutes or one hour.

Figure 2 shows runtimes of the two applications. The overhead of orphan saving is negligible. Likewise, checkpointing has small overhead and the overhead does not seem to depend on the checkpointing interval. This is because our checkpointing is completely asynchronous, both on the nodes sending the data to the coordinator, and on the coordinator itself. Table 1 lists the maximal sizes

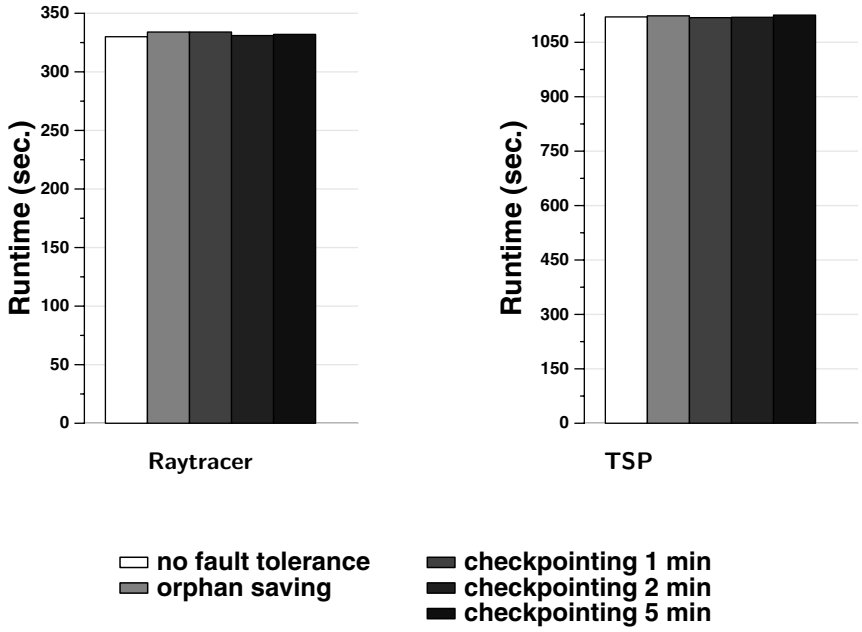

Fig. 2. Application overhead during crash-free execution

Table 1. Checkpoint file sizes

interval	1 min	2 min	5 min
Raytracer	28.0 MB	13.8 MB	16.6 MB
TSP	217 KB	128 KB	55 KB

of the checkpoint files for different checkpoint intervals. The checkpoint files produced by TSP are small, since TSP does not process much data. Raytracer is more data intensive, and therefore produces bigger checkpoint files.

4.2 Performance in the Presence of Crashes

Next, we evaluate the performance of both orphan saving and checkpointing in the presence of crashes. Again, we have run the two applications on 32 nodes in 2 clusters. We removed one of the clusters in the middle of the computation, that is, after half of the time it would take on 2 clusters without processors leaving. The case when half of the processors leave is the most demanding, as the biggest number of orphan jobs is created in this case. On average, the number of orphans does not depend on the moment when processors leave, except for the initial and final phase in the computation.

For our analysis, we compare to two extreme cases. One is *naive fault tolerance*, where orphan results are always discarded. The other extreme is running

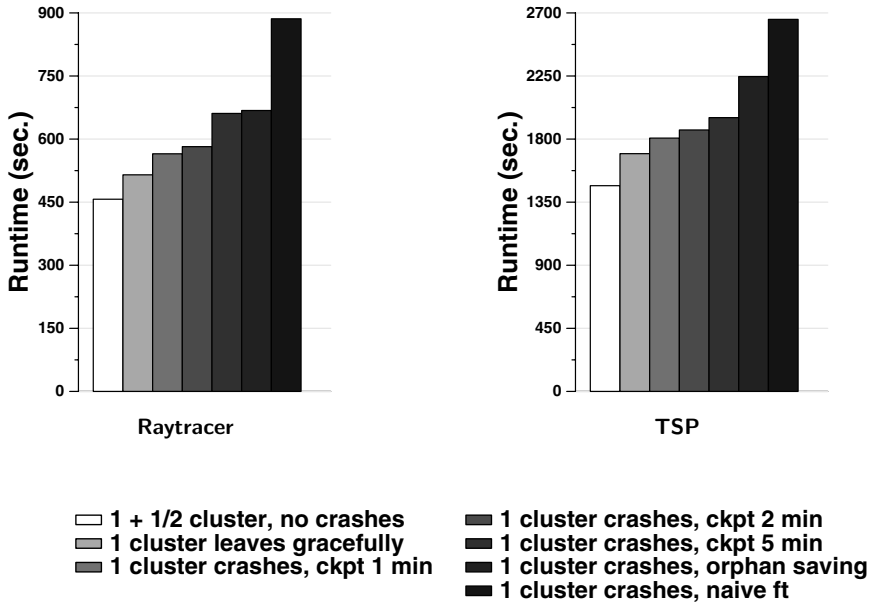

Fig. 3. Application performance in the presence of crashes

on a smaller number of processors right from the beginning, without processors leaving, denoted as the *1+1/2 cluster* case. We also compare to the situation in which nodes are leaving *gracefully* (after a prior notification) where all orphan results are used via the orphan saving mechanism.

We used checkpointing intervals of 1, 2, and 5 minutes. To allow a fair comparison between the checkpointing intervals, we enforced that the crash always occurred exactly in the middle of a checkpointing interval. We achieved this by adjusting the time the *first* checkpoint during the computation was taken.

The graphs in Figure 3 show the runtimes of both applications. The runtimes shown are averages taken over 4 runs. In 50% of the runs, the crashing (or leaving gracefully) cluster contained the master.

Figure 3 shows that orphan saving outperforms the *naive* approach by 15 % to 25 %. Checkpointing improves the performance of the system by further 10 % to 15 %. The performance improvement is the largest with small checkpointing intervals. This is a consequence of the (quasi-) constant checkpointing overhead, as shown in Figure 2. If nodes are leaving gracefully, the orphan saving algorithm provides up to 40% performance improvement over the *naive* algorithm.

Figure 3 also presents runtimes for the situation in which, instead of nodes leaving or crashing, a smaller number of nodes is available to the application right from the beginning. This smaller number of nodes is chosen to be equivalent to the accumulated number of nodes in case of the investigated crashes and graceful leaves, denoted as the *1+1/2 cluster* case. This case is best being compared to the situation of graceful leaves, indicating a small but noticeable cost (12 % for Raytracer and 15 % for TSP) of the orphan saving algorithm which is to be paid

Table 2. Runtimes and checkpoint file size for suspending/resuming applications

application	uninterrupted run	suspend/resume	file size
Raytracer	322 s	360 s	12 MB
TSP	1106 s	1116 s	19 KB

for moving the orphan results and some amount of redundant computation of orphan jobs that get recomputed before their results arrive at the other nodes.

4.3 Performance of Suspending/Resuming an Application

To evaluate the performance of the suspend/resume mechanism, again we ran both applications on 32 nodes in 2 clusters. In the middle of the computation, we stopped the applications, which, in turn, checkpointed their results and exited. Next, we have restarted the applications on the same processor set, using the checkpoint files just created.

The runtimes of both uninterrupted and suspended/resumed execution, along with the checkpoint file sizes, are listed in Table 2. These times are application runtimes only, they do *not* include the overhead of scheduling and (re-)starting the applications. The overhead of suspending and resuming an application is 11% for the data intensive Raytracer application and only 1% for the compute-bound TSP. This overhead is caused by the need to write and read the checkpoint file, The size of the checkpoint file is determined by the checkpoint data structures (including data serialization) and the size of checkpoint intervals. Practically no work is lost while suspending and resuming an application.

5 Related Work

Checkpointing is used in grid computing by systems such as Condor [7] and Cactus [8]. Dynamite [9] uses checkpointing to support load balancing through the migration of tasks for PVM and MPI applications.

Unfortunately, checkpointing causes execution time overhead, even if there are no crashes, mainly caused by writing the state of the processes to stable storage. This overhead might be reduced by using concurrent checkpointing [10]. Another problem of most checkpointing schemes is the complexity of the crash recovery procedure, especially in dynamic and heterogeneous grid environments where rescheduling the application and retrieving and transferring the checkpoint data between nodes is non-trivial. The final problem of checkpointing is that in most existing implementations, the application needs to be restarted at the same number of processors as before the crash, so it does not support malleability. An exception is SRS [11], a library for developing malleable data-parallel applications.

The checkpointing overhead can be reduced by application-level checkpointing, as, e.g., done by Cactus [8]. Here, the application itself determines which data to checkpoint, allowing to reduce the data to a minimum. Satin pushes this idea even further by not only writing small data sets, but also by writing the data asynchronously, without interrupting the ongoing computation. This leads to a very efficient checkpointing scheme, albeit being restricted to divide-and-conquer applications.

Several fault tolerance mechanisms for divide-and-conquer applications have been proposed. In the DIB system [12], processors redo work of other processors even if no crash has been detected. Redoing occurs while a processor waits for its steal request being granted. Instead of staying idle, the processor starts redoing work that was stolen from it earlier but the result of which has not yet been received. This approach is robust since crashes can be handled even without being detected. However, this strategy can lead to a large amount of redundant computation.

Another approach was proposed in [13]. Here, the problem of orphan jobs is partially addressed by storing not only the identifier of the parent processor (the processor from which the job was stolen), but also the identifier of its *grandparent* processor. When the parent processor crashes, the orphaned job is directed to the grandparent instead. Obviously, if both ancestor processors crash, the orphaned job cannot be reused anymore. While this mechanism can be extended further, the price to pay is higher overhead for the additional control data.

Atlas [2] is yet another divide-and-conquer system, based on CilkNOW [14], an extension of Cilk [15], a C-based divide-and-conquer system, to networks of workstations. Atlas was designed with heterogeneity and fault tolerance in mind but aims only at moderate performance. Its fault tolerance mechanism is also based on redoing work. The problem of orphan jobs is not addressed in Atlas.

6 Conclusion

Grid applications need to be fault tolerant, malleable, and migratable. In previous work, we have presented *orphan saving*, an efficient mechanism addressing these issues for divide-and-conquer applications. With orphan saving, applications are guaranteed to complete successfully, as long as at least one processor is alive, at any moment during the execution.

In this paper, we have presented a mechanism for writing partial results to checkpoint files, adding the capability to also tolerate the *total loss* of all processors, and to allow suspending an application and to resume it later, whenever CPU's become available again.

Our performance evaluation has shown that both fault-tolerant mechanisms have only negligible overheads in the absence of faults. This allows us to use very short checkpointing intervals, such as one minute. In the case of faults, the new checkpointing mechanism outperforms orphan saving by 10 % to 15 %. Due to the short checkpointing intervals, our mechanism is recovering from crashes very efficiently. In our tests, suspending and later resuming an application has

only between 1 % and 11 % overhead, compared to uninterrupted runs. We also use a special technique, *checkpoint file compression*, to control the size of the checkpoint file.

Divide-and-conquer lends itself very well for fault-tolerant, malleable, or migratable execution, because any job of an application's execution tree can always be recomputed in case its result was lost. Both our mechanisms, orphan saving and the new checkpointing scheme, can execute very efficiently due to effective re-use of partial (orphan) results in the case of crashes, malleability, or migration.

The new checkpointing mechanism also adds the capability of suspending the execution of an application, and to resume it later from a checkpoint file. Due to the robustness of divide-and-conquer, the computation can even be resumed from (the valid parts of) a damaged checkpoint file which might be the result of a crash of the processor writing the file. With this added value, divide-and-conquer becomes an even more attractive paradigm for implementing grid applications.

Acknowledgements

This work has been partially supported by the *CoreGRID* Network of Excellence, funded by the European Commission's FP6 programme (contract IST-2002-004265). We would like to thank our alumnus student Kris Borg who implemented the first version of the checkpointing mechanism for his M.Sc. thesis project.

References

1. Wrzesinska, G., van Nieuwport, R.V., Maassen, J., Bal, H.E.: Fault-tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. In: 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), IEEE Computer Society Press, Los Alamitos (2005)
2. Baldeschwieler, J., Blumofe, R., Brewer, E.: ATLAS: An Infrastructure for Global Computing. In: Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications, Connemara, Ireland, pp. 165–172 (September 1996)
3. van Nieuwpoort, R.V., Kielmann, T., Bal, H.: Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Snowbird, Utah, USA, pp. 34–43 (June 2001)
4. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Kielmann, T., Bal, H.E.: Satin: Simple and Efficient Java-based Grid Programming. Scalable Computing: Practice and Experience 6(3), 19–32 (2005)
5. Allen, G., Davis, K., Goodale, T., Hutanu, A., Kaiser, H., Kielmann, T., Merzky, A., van Nieuwpoort, R., Reinefeld, A., Schintke, F., Schütt, T., Seidel, E., Ullmer, B.: The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. Proceedings of the IEEE 93(3), 534–550 (2005)
6. Li, K., Naughton, J.F., Plank, J.S.: Real-time, concurrent checkpointing for parallel programs. In: 2nd ACM SIGPLAN Symposium on Principles and Practice of Parall el Programming (PPoPP'90), pp. 79–88. ACM Press, New York (1990)

7. Litzkow, M., Livny, M., Mutka, M.: Condor - A Hunter of Idle Workstations. In: Proceedings of the 8th International Conference of Distributed Computing Systems, San Jose, California, pp. 104–111 (June 1988)
8. Allen, G., Benger, W., Goodale, T., Hege, H.C., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., Shalf, J.: The Cactus Code: A Problem Solving Environment for the Grid. In: Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9), Pittsburgh, Pennsylvania, USA, pp. 253–260 (August 2000)
9. Iskra, K.A., Hendrikse, Z.W., van Albada, G.D., Overeinder, B.J., Sloot, P.M.A., Gehring, J.: Experiments with migration of message-passing tasks. In: Buyya, R., Baker, M. (eds.) GRID 2000. LNCS, vol. 1971, pp. 203–213. Springer, Heidelberg (2000)
10. Plank, J.: Efficient Checkpointing on MIMD architectures. PhD thesis, Princeton University (1993)
11. Vadhiyar, S.S., Dongarra, J.J.: SRS – a framework for developing malleable and migratable parallel applications for distributed systems. Parallel Processing Letters 13(2), 291–312 (2003)
12. Finkel, R., Manber, U.: DIB – A Distributed Implementation of Backtracking. ACM Transactions of Programming Languages and Systems 9(2), 235–256 (1987)
13. Lin, F.C.H., Keller, R.M.: Distributed Recovery in Applicative Systems. In: Proceedings of the 1986 International Conference on Parallel Processing, University Park, PA, USA, pp. 405–412 (August 1986)
14. Blumofe, R., Lisiecki, P.: Adaptive and Reliable Parallel Computing on Networks of Workstations. In: USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems, Anaheim, California, pp. 133–147 (January 1997)
15. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing 37(1), 55–69 (1996)