Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid

Gosia Wrzesińska, Rob V. van Nieuwpoort, Jason Maassen, Henri E. Bal

Dept. of Computer Science
Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{gosia, rob, jason, bal}@cs.vu.nl

Abstract

*Grid applications have to cope with dynamically chang-
ing computing resources as machines may crash or be
claimed by other, higher-priority applications. In this paper,
we propose a mechanism that enables fault-tolerance, mal-
leability (e.g. the ability to cope with a dynamically chang-
ing number of processors) and migration for divide-and-
conquer applications on the Grid. The novelty of our ap-
proach is restructuring the computation tree which elimi-
nates redundant computation and salvages partial results
computed by the processors leaving the computation. This
enables the applications to adapt to dynamically changing
numbers of processors and to migrate the computation with-
out loss of work. Our mechanism is easy to implement and
deploy in grid environment. The overhead it incurrs is close
to zero. We have implemented our mechanism in the Satin
system. We have evaluated the performance of our system
on the DAS-2 wide-are system and on the testbed of the Eu-
ropean GridLab project.*

1. Introduction

In grid environments, computing resources change dy-
namically, as machines may crash or be claimed by other,
higher-priority applications. Grid applications need to be
fault-tolerant and *malleable* [14], that is, able to cope with
increasing and decreasing numbers of processors. In this pa-
per we present a system that supports fault-tolerance, mal-
leability and migration for divide-and-conquer applications.

Divide-and-conquer parallelism is a popular and effec-
tive paradigm for writing parallel grid applications [4, 19].
The divide-and-conquer paradigm is a generalization of the
master-worker paradigm recommended by the Global Grid
Forum as an efficient paradigm for writing grid applica-
tions [15]. The master-worker paradigm typically uses only
one process (the master) to split up work, which restricts
the type of applications that can be implemented with it.
Also, the performance of the master process can be a bottle-
neck of application performance. The hierarchical master-
worker grid system [13] uses two levels: a single supervisor
process controls multiple master processes each with own
set of workers. The divide-and-conquer paradigm is a fur-
ther generalization and allows any computation to be split
up, in a recursive way. Therefore, many applications can be
implemented with divide-and-conquer, including search al-
gorithms (e.g., satisfiability solver, chess), N-body simula-
tions, raytracing and many others.

We present a divide-and-conquer system that can adapt
to dynamically changing numbers of processors. While
adding processors to a divide-and-conquer computation is
straightforward (as will be explained in Section 3), han-
dling leaving machines is non-trivial. Existing divide-and-
conquer systems [4, 6, 10, 16] handle this by recomputing
work done by leaving processors. However, those systems
suffer from much redundant computation which degrades
their performance. First, they do not reuse *orphan* work,
that is, tasks that are dependent on tasks done by leaving
processors. Orphan work is discarded and recomputed. Sec-
ond, existing systems cannot reuse the partial results from
leaving processors.

We propose a recovery mechanism which salvages or-
phan work and reuses partial results computed by the leav-
ing processors, when they leave *gracefully*. This occurs,
for example, when the processor reservation is coming to
an end or when the application receives a notification that
it should vacate part of its processors for another, higher-
priority application. When the processors leave gracefully,
the work done by them is randomly distributed over the
other processors. When they leave unexpectedly (crash), the
work they have done is recomputed, but orphans caused by
this crash are reused. Our mechanism salvages both orphan
work and work done by leaving processors by restructuring
the computation tree. When processors are leaving grace-

fully, our mechanism can save nearly all the work done by the leaving processors. Therefore, we use our technique for efficient *migration* of the computation: to migrate the computation from one cluster to another, we first add the new cluster to the computation and then (gracefully) remove the old one.

Our mechanism is easy to implement and deploy in grid environments, as it does not need specialized services such as stable storage. Implementing stable storage on the Grid is non-trivial, due to the heterogeneous resources and different administrative domains [22]. This makes traditional fault-tolerance techniques, such as checkpointing, difficult to implement in grid environments.

The overhead of our mechanism during crash-free execution is very small. Our mechanism can handle crashes of multiple processors or entire clusters.

We implemented our mechanism in Satin [20], a Java-based divide-and-conquer system designed for grid environments. We evaluated the performance of our mechanism on the wide-area Distributed ASCI Supercomputer (DAS-2) and on the heterogeneous testbed of the European Grid-Lab [2] project.

The rest of this paper is structured as follows. In Section 2, we introduce the Satin divide-and-conquer system. In Section 3, we present our malleability algorithm. We present the performance evaluation of our system in Section 4. In Section 5, we discuss related work. Finally, we draw our conclusions in Section 6.

2. Divide-and-Conquer in Satin

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. An example *execution tree* of a divide-and-conquer application is shown in Figure 1. By allowing subproblems to be divided recursively, the class of divide-and-conquer algorithms subsumes the master-worker algorithms, thus enlarging the set of possible grid applications.

Divide-and-conquer applications can be run efficiently in parallel by running different subproblems (jobs) on different machines [7]. In Satin, the 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. In [18], we presented a new work stealing algorithm, *Cluster-aware Random Stealing* (CRS), designed for cluster-based, wide-area systems. CRS is based on the traditional Random Stealing (RS) algorithm that has been proven to be optimal for homogeneous (single cluster) systems [8]. CRS overlaps inter-

cluster steals with intra-cluster ones. It was shown to perform well in grid environments [19]. The logical next step in making Satin grid-ready is extending it with fault-tolerance, malleability and migration support.

3. Fault tolerance, malleability and migration support

Adding a new machine to a divide-and-conquer computation is straightforward: the new machine simply starts stealing jobs from other machines. Removing processors is not trivial. Existing divide-and-conquer systems [4, 6, 10, 16] handle this by recomputing work stolen by leaving processors. However, those systems suffer from much redundant computation, because they are not able to reuse *orphan jobs*, which will be described in Section 3.2. Those systems are also not able to reuse partial results from leaving processors.

Like in existing systems, our malleability mechanism is based on recomputing jobs stolen by leaving processors. The novelty of our approach, however, is the restructuring of the computation tree to reuse as many already computed partial results as possible. Using this approach we eliminate redundant computation by salvaging orphan jobs. We also reuse partial results from leaving processors, if they leave gracefully.

We assume that when processors are leaving gracefully, the remaining processors are notified about it by the operating system or by the leaving processors themselves. Unexpected leaves (crashes) are detected by the communication layer in our system. Crashes do not have to be detected immediately after they occur, but they must be detected eventually.

3.1. Recomputing jobs stolen by leaving processors

To be able to recompute jobs stolen by leaving processors, we keep track of all the jobs stolen in the system. Each processor maintains a list of jobs stolen from it. For each job, the processorID of the thief is stored along with the information needed to restart the job. When one or more processors are leaving, each of the remaining processors traverses its stolen jobs list and searches for jobs stolen by leaving processors. Such jobs are put back in the work queues of their owners, so they will eventually be recomputed. Figure 2 shows the computation tree from Figure 1 after processor 3 has left. Processor 1 will reinsert job 2, on which processor 3 was working, into its work queue. Each job reinserted into a work queue during recovery procedure is marked as "restarted". Children of "restarted" jobs are also marked as "restarted" when they are spawned.
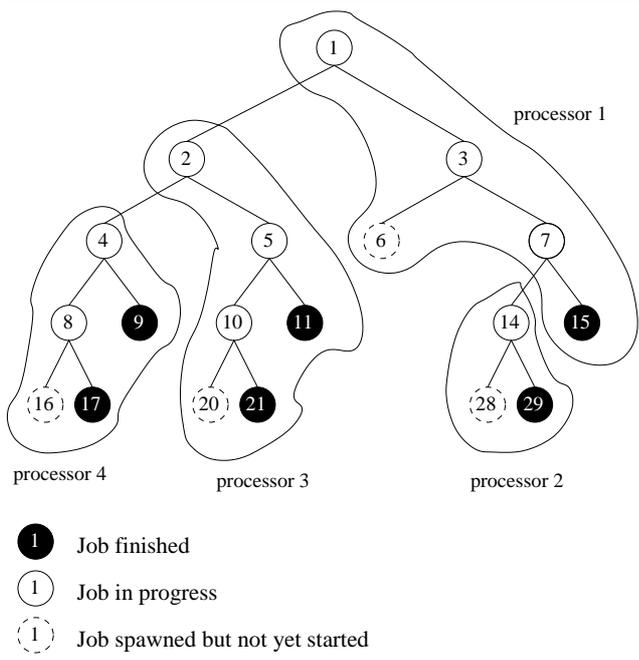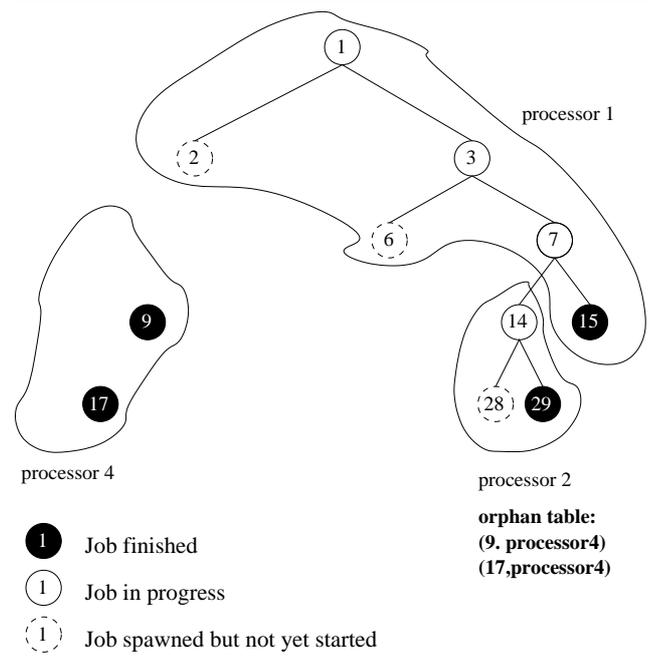
Figure 1. An example computation tree

1 Job finished

1 Job in progress

1 Job spawned but not yet started

processor 1

processor 4

processor 3

processor 2

Figure 3. After the recovery procedure

1 Job finished

1 Job in progress

1 Job spawned but not yet started

processor 1

processor 4

processor 2

orphan table:
(9. processor4)
(17,processor4)
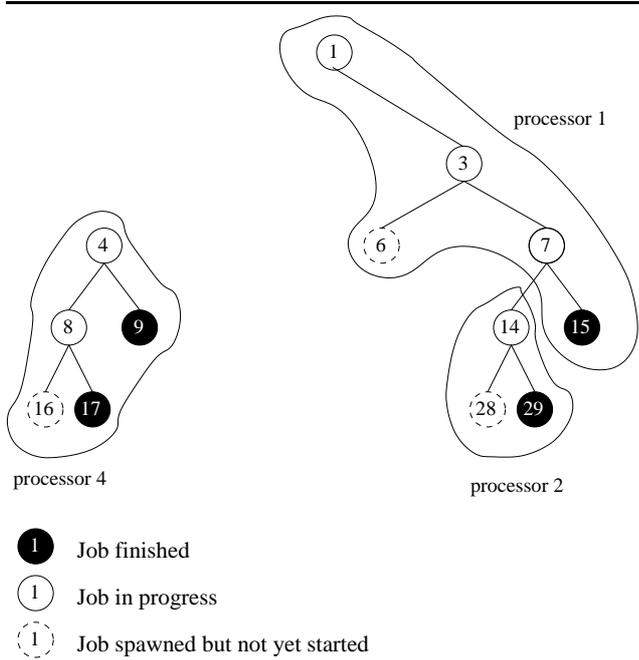
Figure 2. Processor 3 leaves the computation

1 Job finished

1 Job in progress

1 Job spawned but not yet started

processor 1

processor 4

processor 2

Figure 4. Restoring parent-child links

1 Job finished

1 Job in progress

1 Job spawned but not yet started

processor 2

processor 1
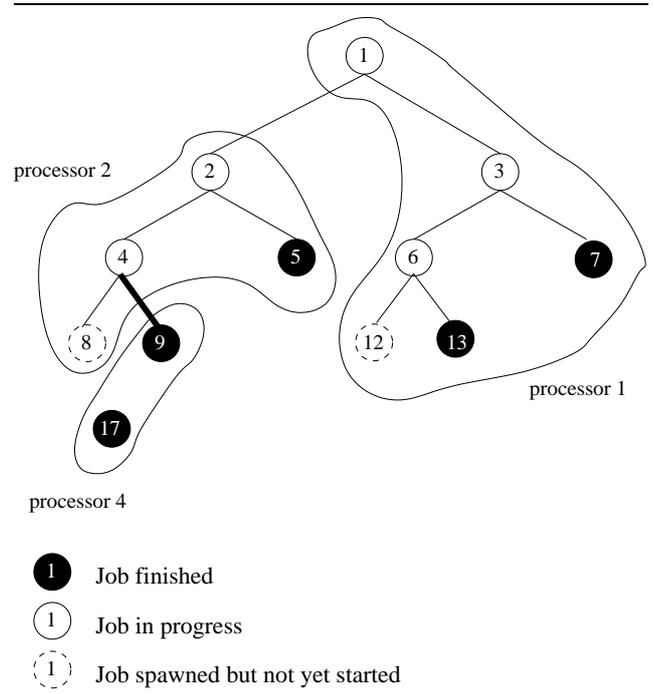
processor 4

3.2. Orphan jobs

Orphan jobs are jobs stolen *from* leaving processors. In Figure 2, job 4 and all its subjobs are orphans. In most existing approaches, the processor which has finished working on an orphan job must discard the result of this job, because it does not know where to return the result. The results of orphan jobs, however, are valid partial results and can be used while recomputing their parents. The results of orphan jobs would be usable if their restarted parents knew where to retrieve them or the orphan task knew the new address to return the result. Thus, salvaging orphan jobs requires creating the link between the orphan and its restarted parent.

We restore links between parents and orphans in the following way: for each *finished* orphan job (jobs 9 and 17 in Figure 2), we asynchronously broadcast a small message containing the jobID of the orphan and the processorID of the processor computing this orphan. We abort the unfinished intermediate nodes of orphan subtrees, since they require little computation: in a typical divide-and-conquer application, the bulk of the computation is done in the leaf nodes, the intermediate nodes only split work and combine the results. The (jobID, processorID) tuples are stored by each processor in a local orphan table. Figure 3 shows the computation tree from Figure 2 after the recovery procedure. Processor 1 put job 2 back in its work queue. Processor 4 aborted the intermediate parts of its orphan subtree and broadcast (jobID, processorID) tuples for jobs 9 and 17. While recomputing jobs which are marked as "restarted", processors perform lookups in their orphan tables for each spawned job. If the jobID of the spawned job corresponds with the jobID of one of the orphans in the table, the processor does not put it in its work queue. Instead, it puts the job in its list of stolen jobs and sends a message to the owner of the orphan requesting the result of the job. On receiving such a request, the owner of the orphan marks it as stolen from the sender of the request. The link between the restarted parent and the orphaned child is now restored. The result of the orphan job is returned and reused. An example is shown in Figure 4. After the recovery procedure the computation was resumed. Processor 2 stole job 2 from processor 1 and started recomputing it. Because job 2 and all its subjobs are marked as "restarted", processor 2 performs a lookup in its local orphan table for each of those jobs. Processor 2 uses the information contained in its orphan table to restore the parent-child link between job 4 and job 9. The result of job 9 will be used while computing job 4. The result of job 17 will be reused in the same way later in the computation.

Note that reusing orphans does not influence the correctness of the algorithm. If the result of an orphan is not found (e.g. because the broadcast message does not arrive in time),

the job can always be recomputed. Reusing orphans only improves the performance of the system. Therefore, broadcasting does not need to be reliable. Scalable broadcasting algorithms, such as *gossiping*, can also be used.

3.3. Partial results on leaving processors

For saving partial results on leaving processors we use the same mechanism as for saving orphan jobs. If a processor knows that it has to leave the computation, it chooses another processor randomly, transfers all the results of the *finished* jobs to the other processor and exits. Those jobs are treated as orphan jobs: the processor receiving the finished jobs broadcasts a (jobID, processorID) tuple containing *its own* processorID for each received result. Next, the normal crash recovery procedure is executed by all the processors that did not leave. The processors that left are treated as crashed processors. The partial results from the crashed processors are linked to the restarted parents. In Figure 1, processor 3 has finished jobs 11 and 21. If it knows that it has to leave, it can send those jobs to one of the remaining processors and the jobs will be used in further computation.
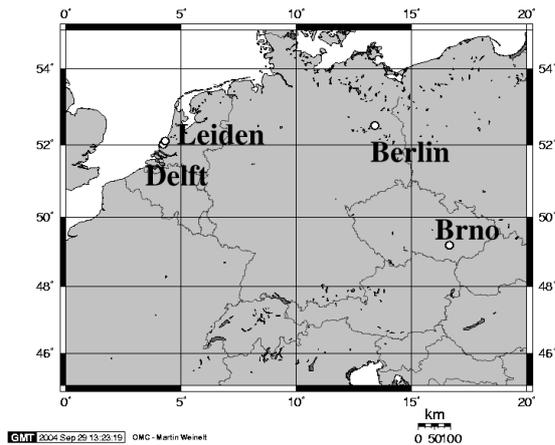
3.4. The master leaving

The leave of the master, that is, the processor which spawned the job which is the root of the execution tree, is a special case. Since the root job was never stolen, it will not be restarted during the normal recovery procedure in which jobs stolen by leaving processors are restarted. When the master leaves, the remaining processors elect the new master which will respawn the root job, thereby restarting the application. The information needed to restart the application is replicated on all processors. The new run of the application will reuse the partial results of the orphan jobs from the previous run (when the master leaves, all jobs become orphans).

3.5. Adding processors

A processor may join the computation *after* the recovery procedure was executed by other processors (e.g., if new processors were added to replace leaving processors). In that case, its orphan table is empty and it has to download an orphan table from one of the other processors, to be able to reuse partial results. It piggybacks orphan table requests on its steal requests until it receives the table.

3.6. Message combining

During the recovery procedure, one (small) broadcast message has to be sent for each orphan and, if the processors leave gracefully, for each finished job computed by a

Figure 5. Locations of the GridLab testbed sites used for the experiments

	Brno	Berlin	Amsterdam	Delft
Brno	-	86	360	441
Berlin	91	-	98	62
Amsterdam	654	87	-	592
Delft	608	85	415	-

Table 1. Available bandwidth in Mbit/s

	Brno	Berlin	Amsterdam	Delft
Brno	-	21	19	19
Berlin	20	-	16	17
Amsterdam	19	16	-	2
Delft	19	17	2	-

Table 2. Latency in ms

leaving processor. To avoid sending a large number of small messages, we use *message combining*. Instead of sending many small messages we combine them into one large message. This reduces the number of messages sent during the recovery procedure to one broadcast message per machine.

3.7. Job identifiers

The job identifiers (jobID) must be both globally unique and *reproducible*: the identifier of a job that is respawned after processor leaves must be the same as it was before the leaves, otherwise the orphaned children cannot be linked correctly to their parents. We create job identifiers in the following way: the root job is assigned ID=1. The child's identifier is computed by multiplying its the identifier of its parent by the maximal branching factor of the computation tree and adding the number of children the same parent generated before. For example, the second child of a job with ID 4 in a tree with branching factor = 2 will have ID = 2 * 4 + 1 = 9. The jobs in the tree in Figures 1–4 are numbered according to this scheme. In most divide-and-conquer applications, the maximal branching factor of the execution tree is known. If it is not known, however, *level stamps* described in [16] can be used: strings of bytes with one byte per tree level.

4. Evaluation

In this section, we will evaluate our mechanism. First, we will show that our mechanism adds little overhead to Satin when no processors are leaving. Second, we will show that our scheme outperforms the traditional approach, which does not save orphans. We will also demonstrate that when

the processors are leaving gracefully, our mechanism can save nearly all the work done by leaving processors. Finally, we will show that our mechanism can be used for efficient migration of the computation.

Part of our experiments were carried out on the DAS-2 homogeneous wide-area system. The DAS-2 system consists of five clusters located at five Dutch universities. One of the clusters consists of 72 nodes, others of 32 nodes. Each node contains a 1 GHz Pentium III processor and runs Red-Hat Linux 7.2. The nodes are connected both by 100 Mbit Ethernet and by Myrinet [9]. In our experiments we used Ethernet.

To demonstrate that our scheme is suitable for real Grid environments, the other part of our experiments was carried out on the testbed of the European GridLab [2] project. DAS-2 is a part of this testbed. To obtain a stable testbed for our experiments, we had to select a relatively small subset of GridLab machines. We used 24 processors in total located in 4 sites all over Europe: 8 processors in Leiden, 8 in Delft (DAS-2 clusters), 4 in Berlin and 4 Brno. The locations of the machines we used are shown in Figure 5. The cluster in Brno contains 2 nodes with 2 Pentium III 700 MHz processors in each of them. The cluster in Berlin consists of 2 nodes; each contains 2 Xeon 1.7 GHz processors. The nodes in Brno and Berlin are running Linux. The available network bandwidth and network latency are shown in Tables 1 and 2. For uploading and starting applications on remote nodes we used software from the Globus Toolkit.

4.1. Overhead during normal execution

We first asses the impact of our malleability mechanism on application performance when no processors are leaving. We ran four applications on the plain Satin system (i.e., without our mechanism) and on the malleable Satin. We used the following applications:
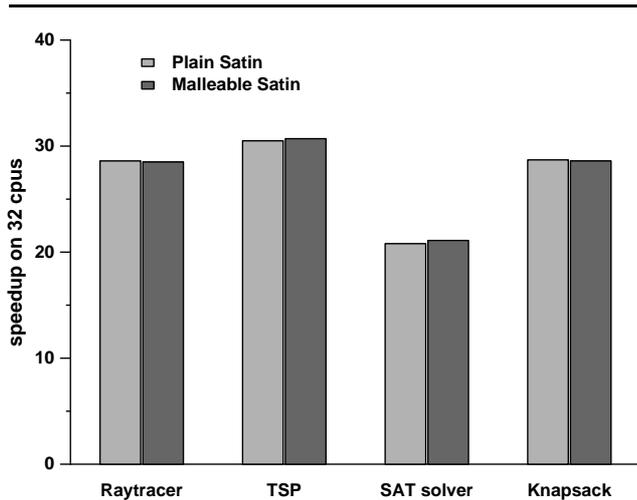
Figure 6. Speedup on 32 processors of a single DAS-2 cluster

- Raytracer which renders a picture using an abstract description of a scene.

- Traveling Salesman Problem (TSP) which has many applications in science and engineering (e.g., manufacturing of circuit boards, analysis of the structure of crystals, clustering of data arrays, etc.).

- Satisfiability Solver (SAT solver) used in industry to verify the correctness of complex digital circuits such as out-of-order execution units in modern processors.

- Knapsack Problem with a wide variety of applications including financial, industrial and sport management, security, aerospace, etc.

Figure 6 shows speedups achieved by the four applications on 32 processors of a single DAS-2 cluster. All speedups were calculated relative to the sequential applications that were run without Satin. The difference in speedup of applications run on plain Satin and on malleable Satin indicates the bookkeeping overhead for the additional data structures of our fault-tolerance mechanism. For all four applications, the difference in speedups in is negligible. This result shows that the overhead added to Satin by our mechanism does not affect the performance of applications.

4.2. The impact of salvaging partial results

In Figures 7 and 8, we show the impact of salvaging orphan jobs and partial results from leaving processors on the performance of the system. In those experiments, we use the Raytracer application since it is sending the most data of all the applications. In the first part of our experiments,

we ran the Raytracer application on 2 DAS-2 clusters (Leiden and Delft) with 16 processors. 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. The number of orphans does not depend on the moment when processors leave, except for the initial and final phase in the computation.

The chart in Figure 7 shows the average runtimes (taken over 4 runs) for the traditional approach (without saving orphans or partial results from leaving processors), our mechanism when processors leave unexpectedly (crash) and our mechanism when processors leave gracefully. We compare this runtime to the runtime on 1.5 clusters (16 processors in one cluster and 8 processors in the other cluster) without processors leaving. The amount of work done by a full cluster during the first half of the computation (before it leaves) is roughly the same as the amount of work done by half of the processors in this cluster without leaving. So, the difference between those two runtimes is the overhead of transferring the partial results from leaving processors and the work we lose because of the leaving processors.

The variation in the runtimes for the traditional algorithm was large: the standard deviation of runtimes was 0.51. This is caused by the fact that the performance of the traditional algorithm depends heavily on the number of orphan jobs created by the leaving processors, as all of those jobs have to be computed twice. Because work is distributed randomly, the variation in the number of created orphans is large which causes a large variation in runtimes for the traditional algorithm. Our algorithm is much less sensitive to the number of orphans, as only small overhead is incurred by reusing this orphan. The standard deviation of runtimes for our algorithm was 0.02.

On average, our algorithm outperforms the traditional approach by 25% in case of unexpected crashes (average was taken over 4 runs). With our approach, the runtime shortens by another 15%, when processors leave gracefully. The difference between the runtime on 2 clusters with 1 leaving gracefully and the runtime on 1.5 clusters is smaller than 8% which shows that we save practically all the work done by the leaving cluster.

We repeated the same set of measurements on four clusters of the GridLab testbed described above. This time we removed one of the four clusters (in Leiden). The results of those tests (Figure 8) are similar to the results on the DAS-2. The differences in runtimes are smaller because the percentage of the processors leaving is smaller.
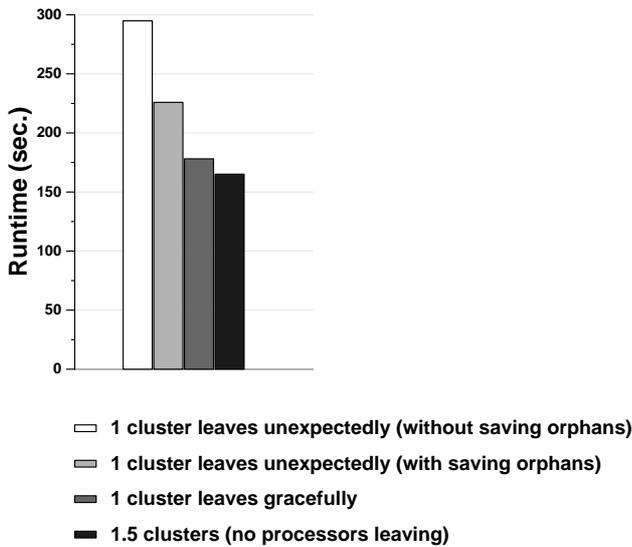
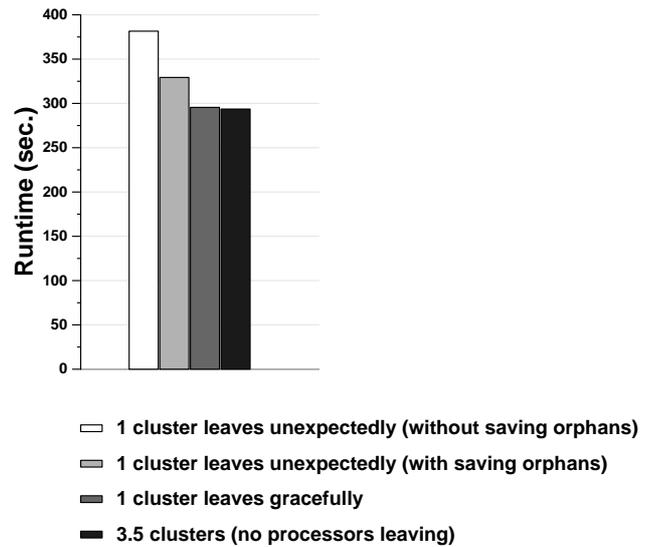**Figure 7. The impact of salvaging partial results – wide-area DAS-2**

Legend:
- 1 cluster leaves unexpectedly (without saving orphans)
- 1 cluster leaves unexpectedly (with saving orphans)
- 1 cluster leaves gracefully
- 1.5 clusters (no processors leaving)

Figure 8. The impact of salvaging partial results – GridLab testbed

Legend:
- 1 cluster leaves unexpectedly (without saving orphans)
- 1 cluster leaves unexpectedly (with saving orphans)
- 1 cluster leaves gracefully
- 3.5 clusters (no processors leaving)

4.3. Migration

In the last experiment, we replaced one cluster with another, that is, we *migrated* a part of the computation. We ran the Raytracer application on the three clusters of the Grid-Lab testbed: in Leiden, Berlin and Brno. In the middle of the computation, we removed gracefully the Leiden cluster and added an identical cluster in Delft. We compared the resulting runtime with the runtime without migration. These runtimes are shown in Figure 9. Because the performance characteristics of the Leiden and Delft clusters are the same, the difference in the runtimes only shows the overhead of migration. With our approach, the overhead is smaller than 2%. This result shows, that our mechanism can be used for efficient migration of the computation.

5. Related work

The most popular fault-tolerance mechanism is *checkpointing*, i.e., periodically saving the state of the application on *stable storage*, usually a hard disk. After a crash, the application is restarted from the last checkpoint rather than from the beginning [3]. Checkpointing is used in grid computing by systems such as Condor [17] and Cactus [1]. Dynamite [12] 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. Writing the state of

the process to stable storage is the main source of this overhead. This overhead might be reduced by using concurrent checkpointing [21]. 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 [23] – a library for developing malleable data-parallel applications. The main advantage of checkpointing is that it is a very general technique which can be applied to any type of parallel applications. Our mechanism is less general – it can be applied only to divide-and-conquer applications – but it supports malleability, is easier to implement and deploy on the Grid and it has smaller overhead.

Several fault-tolerance mechanisms have been designed specifically for divide-and-conquer applications. One example is used in DIB [10], which, like Satin, uses divide-and-conquer parallelism and work stealing. In DIB, when a processor runs out of work, it issues a steal request, but in the mean time it starts redoing (unfinished) work that was stolen from it. This approach is robust since crashes can be handled even without being detected. However, this strategy can lead to much redundant computation. One example is *ancestral-chain* problem: if processor P1 gives work to
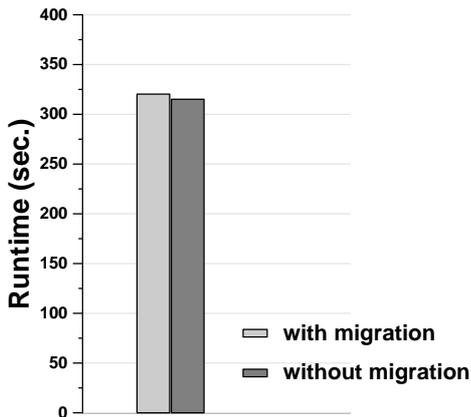
Figure 9. Migration tests results on GridLab testbed

P2, which in turn gives some of it to P3, and P3 crashes, both P1 and P2 will redo the stolen work, so the work stolen by P3 will be redone twice. Furthermore, DIB does not reuse orphan jobs and partial results from leaving processors. Therefore, the amount of redundant computation is very high.

Another approach based on redoing work was proposed by Lin and Keller [16]. When a crash of a processor is detected, the jobs stolen by it are redone by the owners of those jobs, i.e., the processors from whom the jobs were stolen. Orphan jobs are handled as follows. Each job contains not only the identifier of its parent processor (from which the job was stolen), but also the identifier of its *grandparent* processor (from which the parent processor stole the ancestor of our job). When the parent processor crashes, the orphan job is passed after completion to the grandparent processor which in turn passes it to the processor which is redoing the work lost in the crash. The result of an orphan job can thus be reused. However, if both parent and grandparent processor crash, the orphan job cannot be reused anymore. More levels could be used, but that requires storing more data (identifiers). Also, the result of an orphan job is passed to the grandparent processor only after the execution of this job is completed, which may occur a long time after the crash. By that time, some other processor may have already started or even completed redoing the same job. Our experiments show that such situations occur often. Therefore, although this mechanism tries to reuse orphan jobs, it is not efficient especially when multiple crashes occur, and the amount of redundant work is still high. It also does not reuse results from the leaving processors.

Atlas [4] is another divide-and-conquer system. It was designed with heterogeneity and fault tolerance in mind and aims only at reasonable performance. Its fault-tolerance mechanism is also based on redoing the work. Neither orphan jobs nor results from leaving processors are reused in Atlas. Atlas and its fault-tolerance mechanism were based on CilkNOW [6] – an extension of Cilk [7], a C-based divide-and-conquer system. Cilk was designed to run on shared-memory machines while CilkNOW supports networks of workstations.

Fault-tolerance mechanisms based on redoing work lost in crashes have also been proposed for the master-worker programming model. When a worker crashes, the work is redone by another worker. Crashes of the master have to be handled separately. An example of a system that adopts this fault-tolerance mechanism is MW [11] – a programming framework which provides an API for implementing grid-enabled master-worker applications. Charlotte [5] introduces a fault-tolerance mechanism called *eager scheduling*. It reschedules a task to idle processors as long as the task's result has not been returned. Crashes can be handled without the need of detecting them. Assigning a single task to multiple processors also guarantees that a slow processor will not slow down the progress of the whole application. Fault tolerance and malleability for master-worker is inherently simpler than in divide-and-conquer applications: since the task graph is a one-level tree, orphan jobs are not created.

6. Conclusion

In this paper, we presented a mechanism that enables fault-tolerance, malleability and migration for divide-and-conquer applications. We proposed a novel approach to reusing partial results by restructuring the computation tree. Using this approach we minimized the amount of redundant computation which is a problem of many other fault-tolerance mechanisms for divide-and-conquer systems. Our approach also allows to save almost all the work done by the leaving processors, when they leave gracefully. Divide-and-conquer applications using our mechanism can adapt to dynamically changing numbers of processors and migrate the computation between different machines without loss of work.

We implemented our algorithm in Satin, a Java-based divide-and-conquer system. To evaluate our approach, we carried out tests on a wide-area DAS-2 system and in a real grid environment – the European GridLab testbed. In those experiments, we showed that the overhead of our mechanism during normal execution (i.e., without leaving processors) is very small and does not have impact on the performance of applications. We also demonstrated that when processors leave unexpectedly (crash), our mechanism out-

performs the traditional approach (which does not reuse orphans) by 25% when 1 out of 2 clusters crashes. Next, we showed that with our mechanism, applications can handle processors leaving gracefully without loss of work: the overhead is smaller than 8% when 1 out of 2 clusters leaves. Finally, we demonstrated that our mechanism can be used for efficient implementation of migration: the overhead of migrating 1 out of 4 clusters is smaller than 2%.

In earlier work, Satin has been shown to perform excellent in heterogeneous and hierarchical wide-area systems. Our fault-tolerance, malleability and migration support enables Satin applications to efficiently cope with dynamically changing sets of resources and therefore makes it an excellent platform for developing grid applications.

Acknowledgments

References

[1] G. Allen, W. Benger, T. Goodale, G. L. Hans-Christian Hege, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The Cactus Code: A Problem Solving Environment for the Grid. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9)*, pages 253–260, Pittsburgh, Pennsylvania, USA, August 2000.

[2] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor. Enabling Applications on the Grid: A GridLab Overview. *International Journal of High Performance Computing Applications: Special issue on Grid Computing: Infrastructure and Applications*, 17(4):449–466, Winter 2003.

[3] T. Anderson and P. Lee. *Fault Tolerance, Principles and Practice*. Prentice Hall International, 1981.

[4] J. Baldeschwieler, R. Blumofe, and E. Brewer. ATLAS: An Infrastructure for Global Computing. In *Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, pages 165–172, Connemara, Ireland, September 1996.

[5] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proc. of the 9th Int'l Conf. on Parallel and Distributed Computing Systems (PCDS-96)*, pages 181–188, Dijon, France, September 1996.

[6] R. Blumofe and P. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, January 1997.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS'94)*, pages 356–268, Santa Fe, New Mexico, 1994.

[9] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[10] R. Finkel and U. Manber. DIB – A Distributed Implementation of Backtracking. *ACM Transactions of Programming Languages and Systems*, 9(2):235–256, April 1987.

[11] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 43–50, Pittsburgh, Pennsylvania, USA, August 2000.

[12] K. A. Iskra, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, P. M. A. Sloot, and J. Gehring. Experiments with migration of message-passing tasks. In *GRID 2000: The First IEEE/ACM Workshop on Grid Computing*, pages 203–213, Bangalore, India, December 2000.

[13] Y. F. K. Aida, W. Natsume. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *3rd International Symposium on Cluster Computing and the Grid*, pages 156–163, Tokyo, Japan, May 2003.

[14] L. V. Kale, S. Kumar, and J. DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, pages 230–237, Berlin, Germany, May 2002.

[15] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A Grid programming primer. Global Grid Forum, August 2001.

[16] F. C. H. Lin and R. M. Keller. Distributed Recovery in Applicative Systems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 405–412, University Park, PA, USA, August 1986.

[17] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, San Jose, California, June 1988.

[18] R. V. v. Nieuwpoort, T. Kielmann, and H. Bal. Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 34–43, Snowbird, Utah, USA, June 2001.

[19] R. V. v. Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Satin: Simple and Efficient Java-based Grid Programming. In *AGridM 2003 Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, USA, September 2003.

[20] R. V. v. Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, and H. E. Bal. Satin: Simple and efficient java-based

grid programming. *Journal of Parallel and Distributed Computing Practices (accepted for publication)*, 2004.

[21] J. Plank. *Efficient Checkpointing on MIMD architectures*. PhD thesis, Princeton University, 1993.

[22] D. Simmel, T. Kielmann, and N. Stone. GGF Grid Checkpoint Recovery Working Group Charter. Global Grid Forum, January 2004.

[23] S. S. Vadhiyar and J. J. Dongarra. SRS – a framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312, 2003.