Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments

Gosia Wrzesińska, Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, Henri E. Bal
Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{*gosia, rob, jason, kielmann, bal*}*@cs.vu.nl*

http://www.cs.vu.nl/ibis

Abstract

Divide-and-conquer is a well-suited programming paradigm for parallel Grid applications. Our Satin system efficiently schedules the fine-grained tasks of a divide-and-conquer application across multiple clusters in a grid. To accommodate long-running applications, we present a fault-tolerance mechanism for Satin that has negligible overhead during normal execution, while minimizing the amount of redundant work done after a crash of one or more nodes. We study the impact of our fault-tolerance mechanism on application efficiency, both on the Dutch DAS-2 system and using the European testbed of the EC-funded project GridLab.

1 Introduction

Parallel applications are being executed on increasingly large and complex systems like computational grids [1, 4, 24]. In large-scale grids, the probability of a failure is much greater than in traditional parallel systems [15]. Therefore, fault tolerance is becoming a crucial area in grid computing.

In this paper, we propose a fault-tolerance mechanism for divide-and-conquer applications. Divide-and-conquer parallelism is a popular and effective paradigm for writing parallel grid applications [5, 24]. Exploiting the structure of the divide-and-conquer applications allowed us to create a mechanism that has smaller overhead and is simpler to implement than more general techniques such as *checkpointing.*

Divide-and-conquer programs are a generalization of the popular master/worker paradigm. They operate by recursively dividing a problem into subproblems, until they become trivial to solve, and then combining their results. Such programs can be run efficiently in parallel [8]. Also, because of their hierarchical structure, they can be run efficiently on hierarchical grids [24]. An example of a divide-and-conquer system designed for grids is Satin [22], which we use as an experimental platform for our research. Satin is implemented on top of our Java-based grid programming platform Ibis [12, 23]. In Satin, each processor maintains a *work queue* containing jobs (i.e., subproblems) that still need to be executed. The system load is balanced by a special, grid-aware version of work stealing that overlaps local steals with remote ones [22]. Satin is implemented entirely in Java and can run efficiently in distributed, heterogeneous, wide-area environments [24], making it an excellent platform for parallel grid applications.

The divide-and-conquer paradigm has several advantages when implementing fault tolerance. There is no notion of global state in a divide-and-conquer application: function execution does not have side-effects and the result of a function depends only on its input parameters. Function execution will always produce the same outputs if given the same inputs, a property known as *referential transparency* [11]. So, the work lost in a crash of a processor can be redone at any time during execution of the application.

Therefore, it is possible to create a fault-tolerance mechanism based on *redoing* the work lost by processor crashes. Such a mechanism can have very low overhead, as no synchronization between processes is needed and no data needs to be stored on stable storage. Several such techniques have been proposed [5, 7, 14, 18]. However, the common problem of those techniques is *redundant computation* – work done by *live* (i.e., still working) processors that has to be discarded and redone as a result of crashes.

The main contribution of this paper is a novel fault-tolerance mechanism for divide-and-conquer applications that avoids redundant computations by storing partial results in a global (replicated) table. These results can later be reused, thereby minimizing the amount of work lost as a result of a crash. The execution time overhead of our mechanism is close to zero. Our mechanism can handle crashes of multiple processors or entire clusters at the same time. It can also handle crashes of the root node that initially started the parallel computation. We have evaluated the performance of our mechanism on the Dutch wide-area Distributed ASCI Supercomputer 2 (DAS-2), and on the heterogeneous testbed of the European GridLab [2] project. In this experiment, we ran a fault-tolerant parallel application simultaneously on five different parallel machines distributed over Europe.

The rest of this paper is organized as follows: in Section 2, we discuss related work. In Section 3, we present our fault-tolerance mechanism, discuss its correctness and describe its implementation. We present the results of the performance evaluation of our system in Section 4. We conclude in Section 5.

2 Related work

The most popular fault-tolerance mechanism is *rollback recovery*, which comes in two flavours: *checkpoint-based* and *log-based* [13]. With checkpointing, the state of the application is periodically saved 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 [19] and Cactus [1]. Log-based techniques combine checkpointing with logging of nondeterministic events [13]. *Message logging* is a special case of log-based recovery scheme in which all nondeterministic events are modeled as a message receipt. During recovery, the logged events are replayed to the recovering process in their original order. Log-based recovery enables the application to recover beyond its most recent checkpoint and therefore is especially attractive to applications frequently communicating with *outside world* which cannot be rolled back. One of the disadvantages of rollback recovery is its execution time overhead, even if there are no crashes. For checkpointing, the main source of this overhead is writing the state of the process to stable storage. This overhead might be reduced by using concurrent checkpointing [20] and incremental checkpointing [16]. With log-based techniques, copying messages increase communication latency. For systems that implement stable storage by a filesystem available through the network, communication bandwidth required for the application doubles, as each message has to be sent also to the stable storage. Another problem of most rollback-recovery schemes is the complexity of the crash recovery procedure, especially in dynamic and heterogeneous grid environments where rescheduling the job and retrieving and transferring the checkpoint data between nodes is non-trivial. The main advantage of checkpointing is that it is a very general technique which can be applied to any type of parallel applications. The approach we present in this paper is less general, it only applies to divide-and-conquer and master-worker applications, but it has zero overhead when no crashes occur.

Several fault-tolerance mechanisms have been designed specifically for divide-and-conquer applications. One example is used in DIB [14], 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 earlier. 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 the *ancestral-chain* problem: if P1 gives work to 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. Another problem concerns *orphan* jobs, which are stolen *from* a crashed processor. Such a job is computed, but its result cannot be returned to the job's owner since the owner has crashed. The result must therefore be discarded. The same job will be done again while redoing the work given to the crashed processor. Therefore, like in the case of ancestral chains, part of the work will be done twice by live processors. Satin's fault tolerance algorithm is designed to solve both aforementioned problems.

Another approach based on redoing work was proposed by Lin and Keller [18]. 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, the amount of redundant work is still high. The algorithm we propose does not suffer from this problem, because it notifies the grandparent of the orphan job immediately after the crash of the parent.

The likely best-known family of divide-and-conquer systems is the C-based Cilk [8] (for shared-memory machines), and its extensions CilkNOW [7] (for networks of workstations), and Atlas [5]. The latter has been designed with heterogeneity and fault tolerance in mind, but aims only at moderate performance. Its fault-tolerance mechanism is also based on redoing the work. The problem of orphan jobs is not addressed in Atlas.

Fault-tolerance mechanisms have also been proposed for other, similar programming models, such as master-worker. In the master-worker paradigm, all jobs are spawned by a single process, called the master, and distributed among other processes, called workers. The master-worker model can be viewed as a special case of the divide-and-conquer programming style (it is thus less expressive than divide-and-conquer). Therefore, similarly to the divide-and-conquer model, fault tolerance in master-worker systems can be provided by redoing the work lost in a crash. Since only the master processor spawns work and collects results, there is no orphan jobs problem, as long as the master stays alive. Crashes of the master need to be handled separately. An example of a system that adopts this fault-tolerance approach is MW [17]. MW is a programming framework which provides an API for implementing grid-enabled master-worker applications. It also defines an Infrastructure Programming Interface (IPI) such that it can be ported to use various grid software toolkits. Charlotte [6] 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. Satin tries to avoid this duplicate work altogether.

```
0. if (the master crashed)
       elect a new master

1. forall (job stolen by a crashed processor)
       put job back in the work queue

2. forall (descendant of any job
   stolen from a crashed processor)
       if (descendant is finished)
           store the descendant's result
           in the global result table
       else abort the descendant

3. if (the old master crashed
   and I am the new master)
       restart the application
```

Figure 1: The crash recovery procedure for a live processor

3 Fault tolerance using a global result table

Like other fault-tolerance algorithms for divide-and-conquer applications, our approach is based on redoing work lost in crashes. The novelty of our approach, however, is the elimination of a common problem of other solutions – redundant computation, that is, losing and redoing work done by a *live* processor as a result of a crash of another processor. This happens in case of orphan jobs (jobs stolen *from* crashed processors). With the existing algorithms discussed in Section 2, 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 that result to.

To eliminate the problem of orphan jobs, we use a *global result table* – a concept similar to a transposition table [10] used in game solving environments or the table used in tabled execution of logic programs [21]. It is a table accessible to all processors in which results of jobs can be stored. We use this table during the crash recovery procedure for storing the (partial) results of orphan jobs so that other processors can reuse those results. Jobs stored in the table are identified by their parameters.

The global result table is replicated on all processors. Lookups in the table are local operations, and are therefore cheap. The replicas of the table do not have to be strongly consistent (if a processor does not find a job, it can always recompute it), so updates of the table are propagated to other processors asynchronously. Also, updates are infrequent, since we do *not* store all jobs in the global result table, but only *orphan* jobs in the system; these are less than 1 ‰ of all jobs. Finally, for many applications, the amount of data that needs to be broadcast for each job (the parameters and the result) is small – a few bytes. Therefore, the overhead by using the table is small. We will show this experimentally in Section 4.1.

We assume a fail-stop failure model, that is, if a processor fails, it will no longer transmit any valid messages. If a processor comes back after a crash, it receives a new processor identifier and is treated as a new processor. We also assume reliable communication. The crashes of processors are detected by the communication layer. The crashes do not have to be detected immediately after they occur, but they must be detected eventually.

3.1 The algorithm

To be able to redo jobs lost in crashes, each processor maintains a list of jobs stolen from it. For each job, the processor ID of the thief is stored along with all the information needed to restart the job. When a crash of one or more processors is detected, each live processor executes a crash recovery procedure summarized in Figure 1.

In step 1 of this procedure, the list of stolen jobs is searched for jobs stolen by crashed processors. Such jobs are put back in the work queues of their owners, so they will eventually be recomputed. Each job reinserted into a work queue after a crash is marked as *redone.* The *descendants* of a redone job (i.e., children, grandchildren, etc.) are also marked as redone when they are spawned. Before a processor starts computing a *redone* job, it first performs a lookup in the global result table, because the result of this job might already be stored there, as explained below.

In step 2 of Figure 1, the results of orphan jobs are stored in the global result table, so that they can be found and reused by other processors redoing the jobs' parents. To be more specific, instead of waiting until an orphan job is finished and then storing this job's result, we store the results of those parts (subjobs) of the orphan jobs that are already finished at the moment of the crash. We adopted this approach, because computing the whole orphan subtree might take much time and in the meantime some other processor may start working on the same subtree. In that case, the result stored in the table will not be used and the orphan job will be computed twice. Our experiments have shown that this situation is very common.

If the master crashes (i.e., the processor which spawned the job which is the root of the job tree), the remaining live processors elect a new master (step 0 in Figure 1). Afterwards, the crash recovery procedure proceeds normally. At the end of the procedure, the new master restarts the application (step 3 in Figure 1). The information needed to restart the application is replicated on all processors. The new run of the application will reuse the results stored in the global result table during the crash recovery procedure.

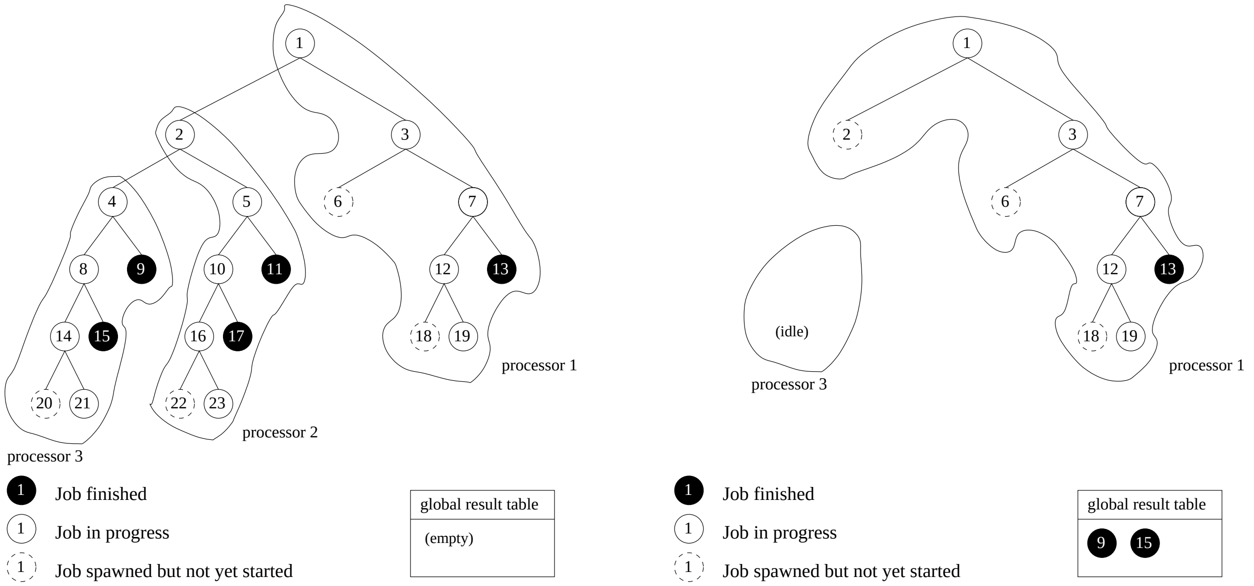
As an example, consider the computation tree shown in Figure 2 (a). The regions delimited by plain lines show how the computation is divided among the processors. When processor 2 crashes all the jobs computed by it are lost (jobs 2, 5, 10, 11, 16, 17, 22, 23). When processor 1 detects the crash, it searches its list of stolen jobs and discovers that job 2 was stolen by a crashed processor. Processor 1 reinserts job 2 into its work queue. Job 4 is an orphan (its parent, job 2, was computed on a crashed machine), so the already completed jobs 9 and 15 are stored in the global result table. Next, the whole orphan subtree (jobs 4, 8, 9, 14, 15, 20 and 21 in Figure 2 (a)) is removed from the system (aborted). After the crash recovery procedure, the computation tree will look as in Figure 2 (b). We discuss the correctness of our algorithm in the appendix.

It may seem that by aborting unfinished jobs, we lose much work. However, as we already explained above, computing the whole orphan subtree and only then storing it in the table would lead to much bigger loss of work, because in the meantime some other processor is likely to start working on the same subtree. In that case the result in the table will not be used and the whole subtree will be computed twice. Besides, the amount of work we actually lose by aborting is small. Most of the unfinished jobs in the system are those which are waiting for results of their subjobs (jobs 4, 8, 14 in Figure 2 (a)). Those jobs are internal nodes of the computation tree. The bulk of the computation, however, is typically done in the leaf nodes. Other aborted jobs are those which were spawned but no processor has started working on them yet (job 20 in Figure 2 (a)). In that case, only a small amount of work, for spawning jobs, is wasted within the runtime system. No application-level work is lost.

Additionally, during normal execution, when a result of a stolen job is returned to the job's owner, it is also stored in the global table. This approach prevents that a job done by one processor is lost in a crash of another processor. This can happen when the result of a job is sent back to a processor which will crash afterwards. Our approach adds only a small performance penalty, since the cost of storing a job in the replicated table is not much higher than the cost of sending it to another processor. The former needs sending one asynchronous broadcast message while the latter needs one asynchronous point-to-point message. Moreover, only a small fraction of the jobs in the system are stolen and sent to another processor. As shown in [22], with typical Satin applications, only one out of 1000–10000 jobs actually gets stolen.

When a processor crashes, we do not lose any work done by any other, live processor, except for the jobs in progress, which need to be aborted. But, as explained above, this causes little work to be lost. In Section 4.2, we will also give an experimental evaluation of this claim.

Note that the use of the global result table does not influence the correctness of the algo-



(a) Before crash of processor 2

(b) After crash of processor 2

Figure 2: An example computation tree

rithm. If the result of a job is not found in the table, the job can always be recomputed. The table only was introduced to improve the performance of the system. Therefore, broadcasting does not need to be reliable. Scalable broadcasting algorithms, such as *gossiping*, can be used. Also, if the size of the table is too large, part of the entries might be safely dropped.

As future work, we plan to extend our algorithm with support for applications with large parameters and results. To avoid broadcasting large amounts of data, the results will be stored locally and only the location from which they can be retrieved is broadcast. Jobs with large parameters are then identified by their position in the job tree rather than their parameters.

3.2 Implementation

We have incorporated our fault-tolerance mechanism in Satin, which is a Java-based divide-and-conquer system. Satin is implemented on top of the Ibis [23] communication library. The core of Ibis is implemented in pure Java, without using any native code. The Satin runtime system and our fault-tolerance extension also are written entirely in Java. The resulting system therefore is highly portable (due to Java's "write once, run anywhere" property) allowing the software to run unmodified on a heterogeneous grid.

The global result table is implemented as a replicated hash table. Keys in this table are records containing the parameters of the job. The hash of such a record is computed as a sum of hashes of all its fields. Hashes of array parameters are computed as sums of hashes of their elements. After an update, an asynchronous update message is broadcast to all replicas of the table. Lookup is a local operation.

4 Evaluation

In this section, we will evaluate our mechanism. We show the following properties of our fault-tolerance mechanism and its implementation in the Satin system:

1. Our fault-tolerance mechanism adds very little overhead to Satin in the absence of crashes;

2. The overhead incurred by lookups in the global result table is very small;

3. With our scheme, little redundant computation is done after a crash;

4. Our scheme outperforms the traditional approach, which does not use a global result table;

5. The scheme is suitable for divide-and-conquer applications that run in a real grid environment.

We show properties 1 and 2 by doing experiments on a single DAS-2 cluster (see Section 4.1). For properties 3 and 4, we do experiments on the wide-area DAS-2 system (see Section 4.2). The DAS-2 system consists of five clusters located at five Dutch universities. Each node contains a 1 GHz Pentium III processor and runs RedHat Linux 7.2. The nodes are connected both by 100 Mbit Ethernet and by Myrinet [9]. In our experiments we used Ethernet. For the experiments on the DAS-2 system, we used the IBM JIT 1.4. The DAS-2 system is used because its homogeneous structure makes it suitable for meaningful performance evaluations. For property 5, we use the testbed of the European GridLab [2] project (see Section 4.3). For the experiments on the GridLab testbed, we used whichever Java implementation was preinstalled on the sites.

4.1 Overhead on normal execution

We first assess the impact of our fault-tolerance mechanism on application performance in the absence of crashes. Therefore, we ran five application kernels with three versions of the Satin runtime system each:

(1) the plain Satin system, without our fault-tolerance mechanism,

(2) fault-tolerant Satin, and

(3) a modified version of fault-tolerant Satin in which lookups in the global result table are performed for *all* jobs.

Comparing (1) and (2) indicates the bookkeeping overhead for the additional data structures of our fault-tolerance mechanism. As there are no crashes, version (2) will not perform any table lookups. Comparing (2) and (3) indicates the overhead of table lookups. The performance of (3) thus gives a lower bound on application performance caused by our fault-tolerance mechanism for those processors of a faulty system that remain alive during an application run.

Figure 3 shows the speedups achieved by our application kernels on 48 processors in all three cases (plain Satin, fault-tolerant Satin, fault-tolerant Satin with lookups for all jobs). The speedups were calculated relative to the sequential Java applications that were run without Satin. For all applications we tested, there is no significant difference in speedup between plain Satin, fault-tolerant Satin and fault-tolerant Satin with lookups for all jobs. We conclude that, in the absence of crashes, our fault-tolerance mechanism incurs only negligible overhead on application performance.

4.2 Efficiency in case of crashes

We will now assess the efficiency of our fault-tolerance mechanism by comparing settings with and without crashes, and with and without using a global result table. For these experiments we use four DAS-2 clusters (Amsterdam, Delft, Leiden, Utrecht), with 16 CPUs each. In preliminary experiments we found that crashes of individual nodes and of whole clusters show similar effects, proportional to the number of crashed nodes. In the following, we will only discuss results with crashes of all nodes in a cluster, as these are more pronounced. Such a crash of a whole cluster represents the situation in which the cluster becomes unreachable due to network problems. From our application kernels we have chosen the knapsack program as it is the most sensitive to crashes of all applications in our preliminary experiments.

Figure 4 shows the results of our assessment. Part (a), on the left side, compares our fault-tolerance algorithm with the traditional ap-

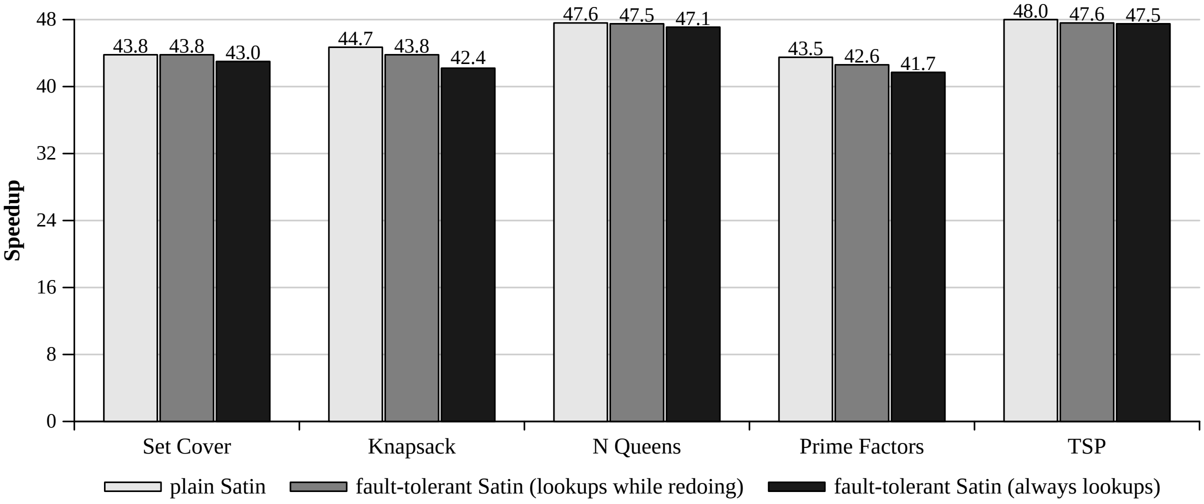


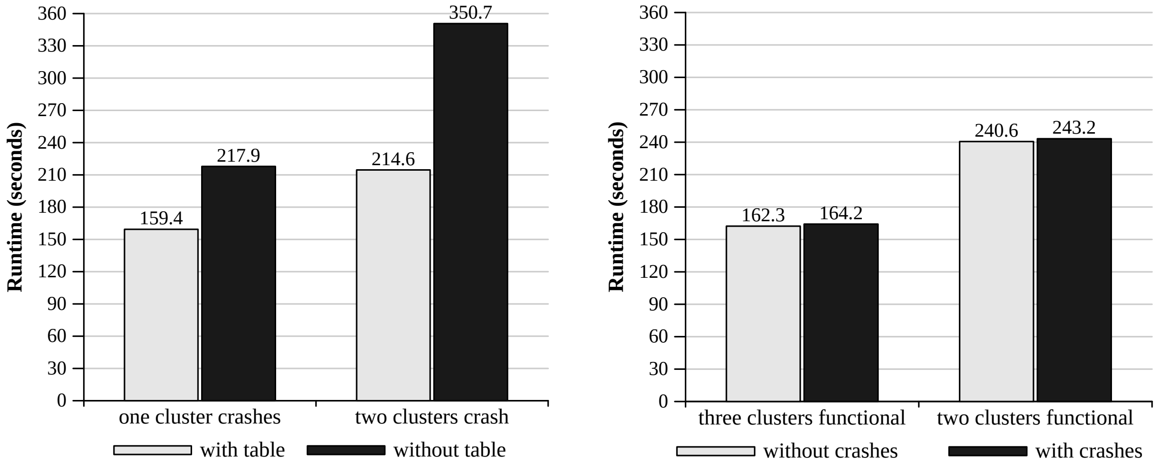
Figure 3: Speedup on 48 processors

proach that does not use a global result table. Traditionally, orphan jobs are simply discarded; work lost in a crash (including discarded orphan jobs) has to be recomputed. This approach is adopted by most other fault-tolerant divide-and-conquer systems.

We first ran the knapsack application on four DAS-2 clusters, without any crashes. The average run time was 125.2 seconds. The left pair of bars in Figure 4(a) shows the application runtimes when one of the four clusters will crash after 50 % of the four-cluster runtime, i.e., after 62.6 seconds. Using our global result table, the application terminated after 159.4 seconds, while not using a result table took 217.9 seconds. Likewise, with two of the four clusters crashing (the right pair of bars), the application terminated after 214.6 seconds with, and after 350.7 seconds without a result table. Clearly, our global result table significantly improves the application runtime in case of crashes.

Figure 4(b), on the right side, shows an assessment of the overhead while handling crashes, caused by redundant computations due to aborting and restarting unfinished parts of orphan jobs. For this experiment we modified slightly the Satin runtime system. With our modified runtime system, we assess the worst case of a processor crash, which is the situation in which none of the results computed by the crashed processor will be propagated to live processors. In our modified system, the to-be-crashed processors will be tagged during application startup. Our runtime system simply discards their results; these results are neither forwarded to other processors, nor written into the result table. The tagged processors are still allowed to steal jobs from other processors and to act as victims of steal requests by other processors. The latter case will generate orphan jobs when the crash occurs. As with Figure 4(a), we trigger a crash of all tagged processors after 62.6 seconds, which is 50 % of a successful four-cluster run.

The left pair of bars in Figure 4(b) compares the runtime of three clusters without crashes (no clusters were tagged as to-be-crashed) with the runtime of four clusters, where after 62.6 seconds one cluster will crash. In both cases, the three live clusters have to compute all jobs of the application. The runtime difference, however, quantifies the amount of additional work, mostly redundant computations, that has to be performed to recover from the crash. As the figure shows, this difference is marginal, e.g. less than 2 seconds. The right pair of bars similarly compares two clusters, with four clusters of which two clusters will crash after 62.6 seconds. Again, the runtime difference is marginal,



(a) impact of result table

(b) impact of crashes

Figure 4: Assessing redundant computation due to crashes with 4 DAS-2 clusters (16 CPUs each)

e.g. less than 3 seconds.

Comparing the bars between parts (a) and (b) of Figure 4 gives two additional insights to the behaviour of Satin in case of crashes. The first observation is that, in the beginning of a run, only few jobs actually get completed with all their children, such that their results can already be returned to other clusters where the jobs had been stolen from. Without crashes, three clusters terminate after 162.3 seconds while four clusters of which one cluster crashes terminate after 159.4 seconds. The marginal gain of 3 seconds is the contribution of the fourth cluster before it crashed. With two clusters crashing, the situation is similar. Two clusters, without crashes, terminate after 240.6 seconds; four clusters with two of them crashing terminate after 214.6 seconds. Here, the contribution of the two crashed clusters is 26 seconds.

The second observation concerns the effectiveness of our global result table. We compare the case of two out of four clusters crashing, without using a result table, (350.7 seconds) with two clusters running without crashes (240.6 seconds). Obviously, without a global result table, handling the crash is more expensive than starting with fewer clusters. This is because orphan jobs are not aborted but computed until they complete, but the results must be discarded and recomputed. However, with our result table, starting with four clusters is beneficial, as the runtime improves from 240.6 seconds to 214.6 seconds. Comparing three clusters with four, out of which one cluster crashes, shows analogous results (comparing 217.9 with 162.3 and 159.4 seconds).

4.3 Grid experiment

To evaluate our divide-and-conquer system in a *real* grid environment, we carried out an experiment on the testbed of the European GridLab project. We used 64 processors in total, using 5 machines located in 3 countries across Europe. Figure 5 shows a map of Europe, annotated with the machine locations. The machines happened to run different flavors of Linux; we used the JVMs installed on the individual sites. The machines are connected by the Internet. The links show typical wide-area behavior, as the physical distance between the sites is large.

We used a real-world application for this test, a satisfiability solver (a classical DPLL SAT solver [26]) implemented with Satin, computing a FPGA design verification problem. On a single node of the Amsterdam DAS-2 cluster, the solver runs for 102,045 seconds, about 28 hours,

Table 1: Machines on the GridLab testbed

location	architecture	operating system	JVM	nodes	CPUs / node	total CPUs
Vrije Universiteit *Amsterdam* *The Netherlands*	Intel Pentium-III 1 GHz	Red Hat Linux 7.2 kernel 2.4.18	IBM 1.4.0	9	2	18
Konrad-Zuse-Zentrum für Informationstechnik *Berlin, Germany*	Intel XEON 2.4 GHz HyperThreading	Red Hat Linux 9 kernel 2.4.20	SUN 1.4.2	1	2	2
Masaryk University *Brno* *Czech Republic*	Intel XEON 2.4 GHz HyperThreading	Debian Linux 3.0 kernel 2.4.27	SUN 1.4.2	2	2	4
Technische Universiteit Delft *Delft* *The Netherlands*	Intel Pentium-III 1 GHz	Red Hat Linux 7.2 kernel 2.4.18	IBM 1.4.0	8	2	16
PC^2 University of Paderborn *Paderborn, Germany*	Intel Pentium-III 850 MHz	Red Hat Linux 7.2 kernel 2.4.7	SUN 1.4.1	12	2	24

in total spawning about one billion jobs.

Figure 6 summarizes the results of our experiment. Using all 5 clusters together, the SAT solver completed after 2494 seconds; using only four clusters (without the Delft cluster), the runtime was 3229 seconds. Then, we deployed our fault-tolerance mechanism: after 50 % of the runtime with five clusters (1247 seconds), we crashed the nodes of the Delft cluster. In this case, the SAT solver terminated after 3082 seconds which is consistent with our tests presented in Section 4.2.

Finally, we used our system to dynamically increase the number of processors at runtime, namely by starting with four clusters, and adding the Delft cluster in the middle of the run, again after 1247 seconds. In this constellation, the SAT solver terminated after 2940 seconds. In both cases, with the crash and with the added cluster, the total processing power is the same. Still, with the added cluster the runtime is 5 % lower than the runtime after the crash. This difference can be attributed to the amount of redundant work performed while recovering from the crash. We can conclude that Satin and its fault-tolerance mechanism can efficiently execute realistic applications in Grid environments while handling changing numbers of available processors.

5 Conclusions

In this paper, we have presented a fault-tolerance mechanism for divide-and-conquer systems. Because our mechanism exploits the properties of the divide-and-conquer paradigm, its overhead during the normal (i.e., crash-free) execution is very small. For all applications we have tested, there is no significant difference in speedup between the application run on the plain Satin system and its fault-tolerant version. We propose a novel approach to salvaging orphan jobs – a global result table. 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. To evaluate our approach, we carried out tests on the Dutch wide-area DAS-2 system and on the GridLab testbed.

Our fault-tolerant Satin system has the potential to become a viable platform for Grid applications. To approach this goal, we are currently working on two extensions. One is an alternative result table mechanism for jobs with large pa-

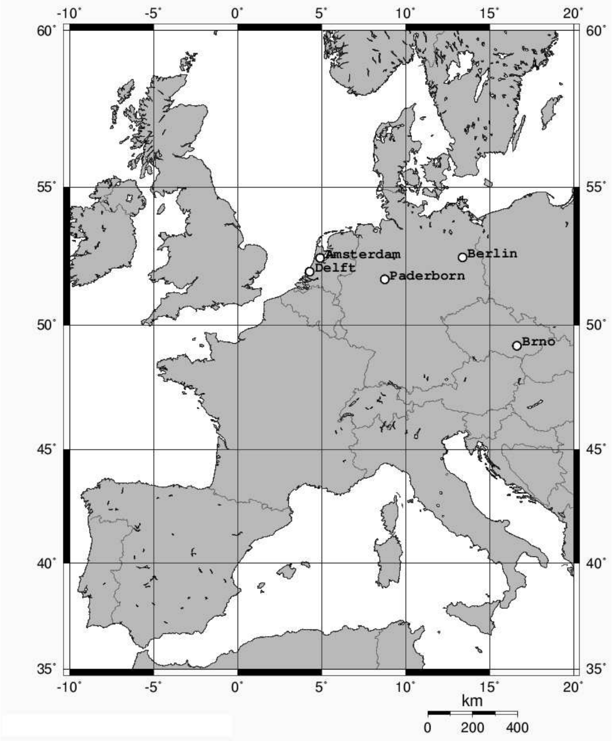


Figure 5: Locations of the GridLab testbed sites used for the experiments

rameter or result variables. The other extension will allow us to use our mechanism for supporting malleability and migration of long-running Satin applications [25].

Acknowledgments

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). Further support came from the European Commission, grant IST-2001-32133 (GridLab), and the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment). We would also like to thank Rutger Hofman, Ceriel Jacobs and Kees Verstoep for their useful comments on this paper.

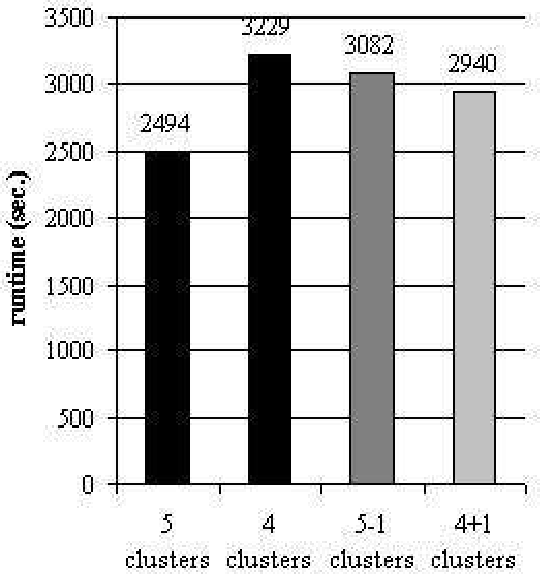


Figure 6: Runtimes of the SAT solver on the GridLab testbed

Kees van Reeuwijk wrote the SAT solver and Matthias Hovestadt helped a lot with accesing the machines at Paderborn University.

References

[1] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, 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] D. Arnold and J. Dongarra. The NetSolve Environment: Progressing Towards the Seamless Grid. In *2000 International Conference on Parallel Processing (ICPP-2000)*, pages 199–206, Toronto, Canada, August 2000.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[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] D. M. Breuker. *Memory versus Search in Games.* PhD thesis, Universiteit Maastricht, 1998.

[11] A. Cherif. *Replication for Fault Tolerant Software Using a Functional and Attribute Grammar Based Computational Model.* PhD thesis, School of Information Science, Japan Advanced Institute of Science and Technology, 1998.

[12] A. Denis, O. Aumage, R. Hofman, K. Verstoep, T. Kielmann, and H. E. Bal. Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems. In *HPDC-13*, pages 97–106, Honolulu, USA, 2004.

[13] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

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

[15] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufman Publishers, Inc., 2004.

[16] S. I. Fredman and C. B. Brown. Igor: A system for program debugging via reversible execution. *ACM SIPLAN Notices, Workshop on Prallel and Distributed Debugging*, 24(1):112–123, 1989.

[17] 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.

[18] F. C. H. Lin and R. M. Keller. Distributed Recovery in Applicative Systems. In *Proceedings of the 1986 International Confer-*

ence on Parallel Processing, pages 405–412, University Park, PA, USA, August 1986.

[19] 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.

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

[21] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings of the 3rd International Conference on Logic Programming*, pages 84–98, London, UK, July 1986.

[22] R. V. van 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.

[23] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.

[24] R. V. van 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.

[25] G. Wrzesińska, R. V. van Nieuwpoort, J. Maassen, and H. E. Bal. Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. submitted for publication, 2004.

[26] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of the 18th International Conference on Automated Deduction*, pages 295–313, Copenhagen, July 2002. ACM.

Appendix: Correctness of the algorithm

A crash recovery algorithm is correct if it brings the system back to an error-free state from which the system can continue to execute. In general, finding such a state is difficult, because the local states of each of the processors must be consistent. In a divide-and-conquer application, however, achieving such a state is easier since the computations performed by each of the processors are to a great extent independent of each other. Jobs can be evaluated independently of each other (due to referential transparency), except that a child must return the result to its parent before the parent can be completed. Therefore, dependencies between processors appear only when one processor steals a job from another processor – the result must be returned before the stolen job's parent can be finished. A crash of a processor thus has a direct impact only on the processors from which the crashed processor stole a job or which stole a job from the crashed processor. To analyze this impact let us assume that processor T (the thief) steals a job from processor V (the victim). A state transition diagram of this process is shown in Figure 7. Let us analyze all possible cases, assuming that neither of the processors is the master.

1. In states 1 and 8 a crash of one processor obviously does not have any influence on the other processor;

2. If V crashes while the job is being stolen from it (states 2 and 3), T cancels the steal request as soon as it detects the crash of V and sends a steal request to another processor;

3. If T crashes while stealing the job, the behavior of V depends on when it detects the crash. If V detects the crash before sending the reply (state 2 or 3), it discards the steal

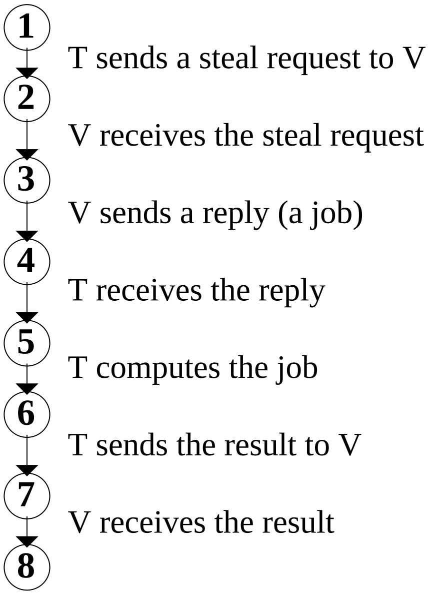


Figure 7: The state transition diagram for stealing, computing and returning the result of a job

request. If V detects the crash after it sent the reply (state 4), it treats the job as stolen and acts as in case 4;

4. If T crashes while working on the stolen job (states 5 and 6), V will not receive the result and will not be able to finish the stolen job's parent. Consequently, all other ancestors of the job (including the root) will not be completed. To solve this problem, V redoes the stolen job (and all its subjobs) after detecting the crash;

5. If V crashes after T stole the job from it (states 4, 5 and 6), the job becomes an orphan. T aborts the job and all its descendents to avoid unnecessary work. The job will be redone while redoing jobs stored on the crashed V (by processors from which V was stealing). The finished parts of the job are stored in the global result table;

6. If T crashes before the result is received (state 7), V acts as in case 4 and redoes the job. However, since the result is sent over the network, it was put in the global result table by T. The result can thus be reused while recomputing the job;

7. If V crashes before it receives the result (state 7), T ignores it. As in case 6, the result will be stored in the global result table and used while redoing the job (by the processor from which an ancestor of this job was stolen by V).

In all cases, live processors will be able to continue their work. Their jobs that have become orphans will be aborted. All other jobs will be completed and their results will eventually be returned to the nodes from which the jobs had been stolen. In particular, the master will be able to successfully compute all its jobs including the root job which marks a successful end of the computation. A special case is the crash of the master. The new master will restart the whole computation.