Fault-Tolerant Static Scheduling for Grids

Bernhard Fechner, Udo Hönig, Jörg Keller, and Wolfram Schiffmann
FernUniversität in Hagen
Department of Mathematics and Computer Science
58084 Hagen, Germany
⟨Bernhard.Fechner,Udo.Hoenig,Joerg.Keller,Wolfram.Schiffmann⟩@fernuni-hagen.de

Abstract

While fault-tolerance is desirable for grid applications because of the distributed and dynamic nature of grid resources, it has seldom been considered in static scheduling. We present a fault-tolerant static scheduler for grid applications that uses task duplication and combines the advantages of static scheduling, namely no overhead for the fault-free case, and of dynamic scheduling, namely low overhead in case of a fault. We also give preliminary experimental results on our scheme.

1. Introduction

Grid computing organizes geographically distributed computing resources into a single system in order to provide performance beyond that typically available at a single site. The sites involved are not under a central control but belong to different organizations. Hence, the resources available in a grid vary dynamically, either by fault or by shutdown. As a consequence, a multitude of fault-tolerance measures for grid computing has been proposed in the literature.

We assume a grid to be a collection of p computing resources, i.e. processing nodes. Each resource may fail with a certain probability f. As in [3, 4, 5, 14] we consider crash faults, i.e. a processing unit that fails is silent from then on. As in [9], we assume that faults can occur at any point in time. A fault can be detected either because a processing unit does not send data at a predetermined time, or because it does not react to a ping request. We assume that system components, both hardware and software obey the fail-stop [11] failure mode and that nodes cannot recover from faults. Therefore, a restart is excluded. In accordance with the crash fault model failures of processors are assumed to be independent from each other [13]. While dependent failures of multiple homogeneous processors may occur as a consequence of an attack, we note that we want to cover failures from faults and not from attacks.

A grid application or job consists of a set of tasks, each task being a sequential program. If one task requires data that have to be computed by another task first, there exists a data dependency between theses tasks, preventing their parallel processing. The dependency structure of a parallel program can be described by means of a directed acyclic graph (DAG), also known as *task graph*. Scheduling a task graph consists of ordering the tasks and mapping them to processing units. Scheduling can occur prior to execution, assuming a fixed state of the grid engine (*static scheduling*), or can be done during execution (*dynamic scheduling*). Providing fault-tolerance can be achieved in several ways: either, when detecting that a processing unit has crashed during execution of a task, the state of this task is recovered from some checkpoint, and this task is re-executed on another processing unit, and all tasks (potentially) to be executed on this processing unit are re-scheduled to another processor. Or, all tasks are executed in multiple instances on several processors, and the grid job can complete as long as one copy of each task survives. The former measure better corresponds to dynamic scheduling, the latter better to static scheduling.

Both ways have their advantages and disadvantages. Dynamic scheduling can adapt to varying resources. Yet to survive crashed tasks, checkpointing is necessary which means overhead in the fault-free case. Also, scheduling and re-scheduling tasks during runtime produces further overhead. Finally, dynamic schedulers typically produce mappings inferior to static schedules as long as the resources remain constant. Static scheduling with task duplication avoids the overheads mentioned above. However, adaptation to varying resources must be handled like crashes of idle processors, and task duplication at least doubles the load on the grid engine. Yet, it has been shown [6, 7] that the presence of data dependencies in task graphs reduces the efficiency of the available resources' usage. Thus, processing units remain idle for a considerable fraction of the job's runtime, so that task duplication hurts less than one would expect. Also, task duplication has already been employed [2, 10] to trade communication for computation, so that using a static scheduler with task duplication already provides some prerequisite for fault-tolerance.

We incorporate fault-tolerance mechanisms into a static scheduler, with the help of task duplication. Yet in contrast to previous work our aim is to minimize overhead in a fault-free execution, because the overwhelming majority of cases will execute without faults. In case of a fault, our aim is to provide a continuation of the computation with a graceful degradation of performance.

The remainder of this article is organized as follows. In Section 2, we review related work on fault-tolerant scheduling of grid applications. In Section 3, we present our fault-tolerance extensions, and the preliminary results of our experimental evaluation. In Section 4, we illustrate variants of our approach by an example. In Section 5, we summarize and give an outlook on future work.

2. Task Scheduling and Fault-Tolerance

In [14] fine-grained, dynamic fault-tolerant scheduling is introduced. Here, redundant computations are avoided by storing partial results in a replicated global table. In GridTS [3, 4, 5] the resources select the tasks they want to execute. This is contrary to the commonly used single resource manager, where the scheduler is responsible in finding the appropriate resources. Favarim et al. assume crash faults. The loss of a task is omitted by using transactions; checkpointing mechanisms are applied to limit the amount of work lost in case of a fault; replication is used to ensure the availability of the tuple space, a shared memory object supporting communication decoupled in time and space so that processes do not need to know each others location or address. In [8] an algorithm to solve branch-and-bound problems is introduced. The algorithm is based on epidemic communication and designed to dynamically react on crash faults to ensure reliability. Within SPHINX [12] another dynamic scheduling algorithm is proposed. It is based on re-scheduling if one or more sites crash.

A static scheduling scheme is proposed by Abawajy [1]. To imply fault-tolerance, tasks are replicated. Yet, this scheme leads to overhead in the fault-free case. Still, we consider this to be the related work closest to our own approach.

If a processing unit crashes, then the work it would have to carry in the future must be distributed onto other processing units[1]. Here we have to distinguish whether spare processing units are available or not. If there is one spare processor in the dynamic setting, then we still have p processing units available, and after the re-execution of the crashed task, the grid job can continue as if nothing had happened. If there is no spare processor in the dynamic setting, then the remaining work must be distributed onto $p-1$ processing units[2] and the remaining runtime increases by a factor

$$\alpha_{dyn,max} \leq \frac{p}{p-1} \; .$$

In practice however, the processor efficiency increases with decreasing processor count, so that we can sharpen the inequality above to

$$\alpha_{dyn,max} \leq \frac{p}{p-1} \cdot \frac{\text{eff}(p)}{\text{eff}(p-1)} < \frac{p}{p-1} \; . \qquad (1)$$

Note that sometimes spare processors may come for free, if a grid of a predetermined size is allocated for a grid job, but the dependencies prevent the scheduler from using all processing units in the grid. A major reason for this contra-intuitive behaviour is communication cost. We will explain this with the help of Fig. 1. Part (a) of this figure shows a simple task graph. Nodes 0 and 5 are artificial nodes that serve to generate unique source and sink nodes in the task graph. Therefore their runtime (rt) is 0, as are the communication times on their outgoing/incoming edges. Part (b)

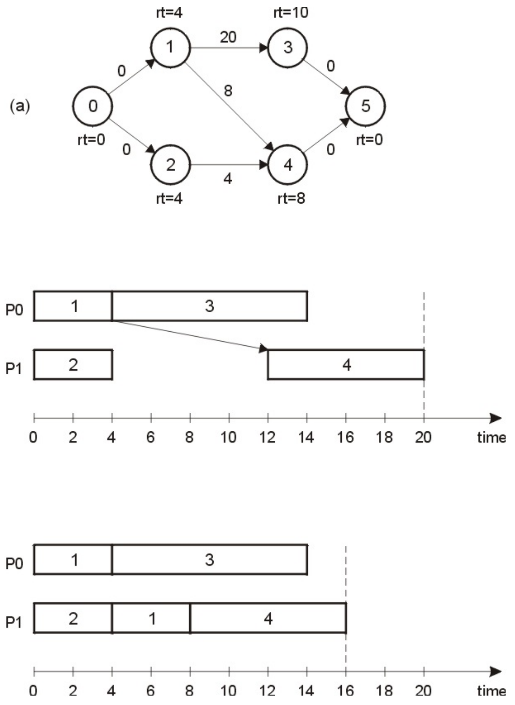

Figure 1. Task Duplication: (a) Task graph, (b) Schedule without duplication, (c) Schedule with duplication. rt=runtime.

[1]We will present our approach with homogeneous processors which may be considered to represent a cluster within a grid. Yet, our approach also extends to time-invariant heterogeneous grid systems.

[2]We assume here that the remaining work can be balanced on $p-1$ processors to the same extent as on p processors.

shows a schedule for this task graph on 2 processing units. Tasks 1 and 3 are scheduled on processing unit P0, and task 3 can start immediately after task 1 has finished, despite an edge $(1, 3)$ with weight 20. This modeling is motivated by the argument that data transfer within one processing unit happens via memory or hard disk, and is basically free compared to communication over a network. In contrast, task 4 can only be started on P1 8 time units after the completion of task 1 because of the communication cost. This kind of behaviour also motivated task duplication in static scheduling. Part (c) of Fig. 1 shows a schedule where two copies of task 1 are executed on P0 and P1. Because the runtime of task 1 is smaller than the communication cost on edge $(1, 4)$, task 4 can now be started earlier, and the *schedule length*, i.e. the time to execute the schedule completely, reduces from 20 to 16 time units.

If all tasks were duplicated, then the crash of a single processor can be overcome in a static schedule, because at least one copy of each task will survive. One would need p spare processors to guarantee a continuation in a static schedule after a fault without performance loss. Each of the spare processors executes exactly the same tasks as one of the p original processors, so that the task graph is executed twice on different hardware. No matter which processor fails, one of the copies of the task graph will be executed without disturbance. More general, if $p + k$ processing units are available, for $0 \leq k < p$, then the schedule length increases by a factor

$$\alpha_{stat,max} \leq \frac{2p}{p+k} \cdot \frac{\text{eff}(2p)}{\text{eff}(p+k)} < \frac{2p}{p+k} \ .$$

Note that, while in the dynamic case only the remaining runtime was increased, here the complete schedule length is increased, as from the start of the grid job on, the tasks have to be scheduled in two copies each on $p+k$ processing units. For $k = 0$, i.e. p processing units, the schedule length increases by a factor of

$$2 \cdot \frac{\text{eff}(2p)}{\text{eff}(p)} \ .$$

While this looks like a serious loss of performance, our experiments indicate that in practice the loss of performance is only around 10% even for $p = 4$, because the efficiency is often quite low due to data dependencies, so that both copies of the task graph can be interleaved. This means that the second copy of the task graph can be scheduled almost completely in the processors' idle gaps that remain after scheduling the first copy of the task graph. This is what [1] does. As the schedule length is increased, the overhead in the fault-free and faulty case is identical.

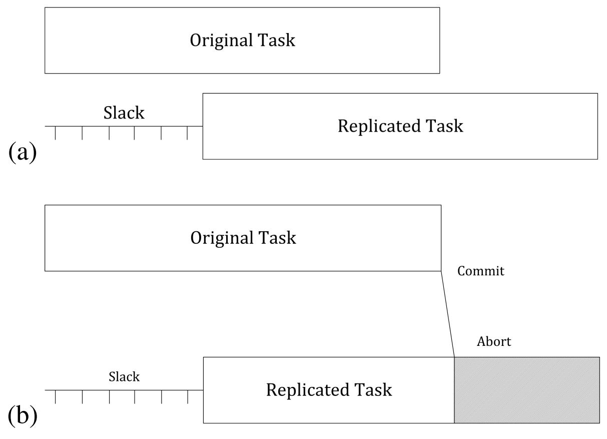

Figure 2. Abort of task copy.

3. Efficient Fault-Tolerant Static Scheduling

Our goal is to provide a static schedule that allows to survive a processor fault but avoids overhead in the fault-free case. We still want to minimize the overhead in the case of a fault, without providing spare processors. Thus, our approach is more efficient than previous schemes. The problem of the state of the art is, that the duplicate of a task does not fit completely in the idle gap of another processor, and thus leads to overhead. We note that if the task itself completes without a fault, then the duplicate is not needed anymore, and can be aborted. This simple difference is illustrated in Fig. 2. The duplicate starts some time after the task itself, which we call *slack*. In part (a) of the figure, the duplicate also terminates later than the task itself by this amount of time. In part (b), the task signals its termination to the duplicate, which leads to an abort of the duplicate a short time after the task itself has terminated. Race conditions between the abort signal and the duplicate's completion can be avoided by choosing a slack that is longer than the maximum communication time of the termination signal.

Hence, we propose to first provide a schedule for the task graph itself, and map a copy of each task onto another processor in an idle gap, but adapt the runtime of this copy such that the copy terminates at the same time that the task itself terminates. In the extreme case that a gap on another processor is only available after the task already terminated, the copy is scheduled into this gap with runtime 0, because in the fault-free case it will never start at all.

Note that when we schedule the task graph with a scheduler that applies task duplication for performance reasons, we only have to place duplicates of tasks that have not been duplicated in the course of the original scheduling, and that only those latter tasks must and may be aborted!

It is obvious that in the fault-free case we experience no overhead, if we neglect the short times that the abort messages need to be communicated. In the case of a fault, we will however experience a delay, because in this case, the copies of the crashed task, and of all tasks to be executed by the crashed processor in the future, will not be aborted, but run to their completion. Thus, we have a situation as in the dynamic case, where the remaining tasks are to be executed on $p - 1$ processors, and the remaining runtime is longer by the factor as given in Eq. (1).

Our method thus combines the advantage of dynamic scheduling in the case of a fault with respect to a small increase in schedule length, and the advantages of static scheduling in the fault-free case: no overhead and more efficient schedule.

So far, we have only considered the mapping of tasks, but not the communication of data. Upon termination, a task sends its results to the dependent task as defined in the task graph, and to the copy of that task. Thus, network traffic is doubled, but only in the sense that each message now has two receiving parties. As it can never happen that both copies of a task terminate (race conditions are avoided by a suitable slack), the number of messages sent is not changed. Note that it cannot be excluded that the first copy of a task is placed on a different processor than its predecessor, and thus experiences communication latency, while the second copy is placed on the same processor as the predecessor, and thus experiences no communication latency. In this case, the second copy would start before the first, and thus terminate before the first. Hence, this case has to be avoided by the scheduler.

Up to now, we did not integrate our approach in a static scheduler. It has been shown in our previous work [6, 7] that an increasing number of available processing elements leads to a decreasing efficiency of the target architecture when processing only one parallel programm at a time. By decreasing the system's size, the execution of the program's tasks is concentrated on a smaller number of processing elements, resulting in a reduced need for inter-processor communication implying fewer and smaller gaps. For this reason, it can be expected that the overhead of our approach in case of a fault increases with decreasing system size. The maximum overhead occurs when the target architecture consists of only two processing elements of which one fails immediately at the beginning, and where the efficiency of the original schedule is exactly 100%. Here, the runtime is doubled, because all tasks are processed sequentially on the single remaining processing element.

In order to test the viability of our scheduling approach for larger target architectures, we did the following preliminary experiment: We scheduled a task graph with 236 tasks on a varying number of processors and duplicated each task with a slack of three time units. This setting represents a kind of worst-case, because in case of a single processor fault, only a subset of the tasks' duplicates (the duplicates of the tasks that were mapped to the faulty processors) have to be processed until completion, even when the fault happens right at the beginning. With 32 shared processing elements and 2×236 tasks, the schedule length elongated by only 58 time units (from 1898 to 1956) when all copies were scheduled with their full length. Thus the overhead in the case of a fault is very small. The runtime increases by a factor of 1.0306, which corresponds to the factor in Eq. (1) for $p = 32$.

In addition to the number of available processing elements, which is an exclusive property of the target architecture, some properties of the task graph also show a significant impact on the efficiency of the target architecture's use as well. It was already shown [6, 7] that a low meshing degree and low communication costs improve schedule efficiency. While a low meshing degree implies a lower number of data dependencies and therefore a reduced chance of time consuming data transfers between processing elements, low communication costs mean shorter communication times and smaller gaps within the resulting schedule. Such schedules leave less computation time unused, implying an increased likelyhood of a prolonged runtime in the case of task duplication. In contrast, task graphs with a high meshing degree or high communication costs will probably result in less efficient schedules with several large gaps, that can be used to host the original tasks' duplicates. Since these duplicates will not affect the schedule length at all, the overhead of our fault-tolerant approach is limited in this case.

Note that by using $k > 1$ copies on different processors, failures of up to k processors can be covered in a similar manner.

4. Three Example Variants of Fault-Tolerant Scheduling

In this section we will illustrate variants of our approach by an example. The considered task graph is given on the left side of Fig. 3. For simplicity, the communication times are set to zero. Here, we want to investigate three special cases of fault tolerant scheduling in more detail. They are characterized by an increasing rate of the gap's utilization for fault tolerant processing.

Note that we can distinguish between two kinds of duplicates: dummy duplicates and (normal) duplicates. A dummy duplicate is a placeholder for a duplicate that will be executed only if the processor for the original task crashed. As already mentioned, we suppose that dummy duplicates (DD<task>) do not consume processing time in the fault-free case. Thus, they can be scheduled not only into a gap but also between two tasks of a static schedule. In contrast, a duplicate (D<task>) will always be executed for some

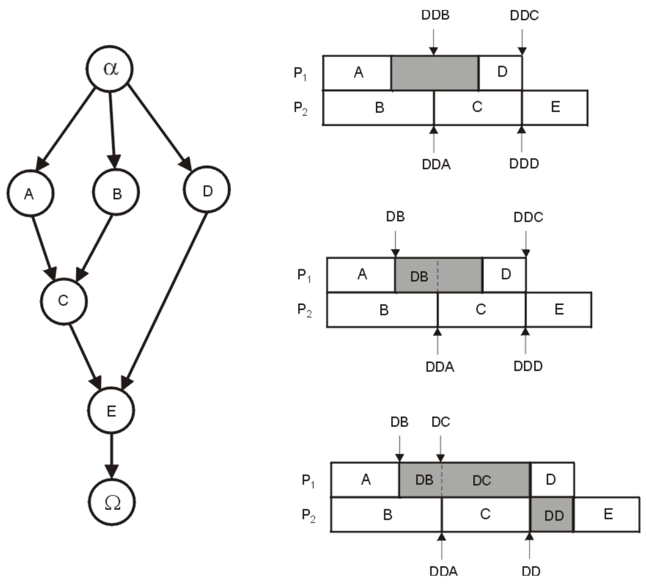

Figure 3. An example task graph and three variants of fault-tolerant schedules.

time (or even completely) during a gap in the schedule. As we have already seen, a duplicate can be aborted if the corresponding original task terminated correctly. Besides, it may also be necessary to stretch the gap if the end of the original task is located behind the end of the gap. In this case the schedule length will be increased as well.

According to those special cases we construct three different fault-tolerant schedules. On the upper right side of Fig. 3, you see a schedule which only makes use of dummy duplicates. These are either scheduled at the end of tasks or during the gray shaded gap (see DDB). In this case, the dummy duplicates ensure that the task graph will be executed correctly (with reduced performance) even if one of the processors fails. Nevertheless, the available processing power of the gap is not exploited.

This is changed on the middle right by converting a dummy duplicate (DDB) to a duplicate (DB) which can be started at the end of task A on processor P_1. If P_2 does not fail DB will be aborted at the end of task B and the rest of the gap is left unused. Cancelation can be accomplished by run-time daemons for each processor. These daemons exchange information about task completion. If P_2's daemon reports the successful completion of task B to P_1's daemon it will shut down DB because it is no longer useful to execute the duplicate.

Finally, in the lower right of Fig. 3 a duplicate of task C is scheduled at the beginning of the still unused part of the gap. Because at the ordinary end of that gap it cannot be decided if the original task will terminate correctly, in this case we have to extend the duplicate until the end of C. As a consequence, we also have to shift the rest of the schedule (simultaneously on both processors) by the amount of the delay of the gap. Here, it corresponds to a delay of the length of task D. Thus, in this case the schedule length will be increased and an additional new gap on processor P_2 appears. Fortunately, as seen in the lower right of Fig. 3, this new gap can now be used for a duplicate of task D. Thus, even if the total schedule length will be increased new opportunities arise to use emerging gaps for fault tolerant processing.

In general, we can choose between the three strategies described above. While in the first case idle processors are not used at all we can gradually increase the efficiency by the other two variants. Even though the highest efficiency will be observed in the last case we have to accept an increase in the schedule length. On the other hand we will get faster reaction if a processor fails because more fault tolerant processing takes place simultaneously.

In this section, only a two processor schedule was considered for simplicity. Usually, for a specific schedule there is a larger number of processors. Thus, we also have to decide between multiple duplicates for filling up a gap. In order to select a task for (normal) duplication, a selection criterion is required. As seen in the examples above, one can use the degree of the overlap between the original task and the gap as criterion function. If we want to maximize the efficiency the task with the greatest overlap should be selected as a duplicate and for the remaining tasks dummy duplicates are created. Moreover, depending on the desired gap's utilization, we can use one of the last two variants.

5. Conclusions

We presented a novel static scheduling algorithm for grid applications that uses task duplication to allow a grid job to survive processor failures. The scheduling algorithm combines the advantages of static scheduling with respect to better schedule efficiency and avoidance of overhead in the fault-free case, with the advantage of dynamic scheduling in the case of a fault, i.e. low increase of schedule length. We have presented some very preliminary experimental results that support our theoretical analysis. Our future work will consist of a full implementation of this scheduler and further explorations of the extensions and trade-offs that so far are only sketched and not fully developed. Instead of insisting on no overhead in the fault-free case, it is also possible to allow a small amount of overhead in the fault-free case, while improving the situation in the case of a fault. For example, one might schedule the second copy of each task with a fixed fraction of the task's original runtime. Also, one might control and even fix as far as possible the amount of slack between both copies of a task. Finally one could use a static scheduler with task duplication in order to reduce the number of tasks to be duplicated purely for fault-tolerance reasons.

References

[1] J. H. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *Proc. 14th IPDPS*, page 238b, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[2] S. Darbha and D. P. Agrawal. A task duplication based optimal scheduling algorithm for variable execution time tasks. In *ICPP*, pages 52–56, 1994.

[3] F. Favarim, J. da Silva Fraga, L. C. Lung, M. Correia, and J. F. Santos. Exploiting tuple spaces to provide fault-tolerant scheduling on computational grids. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 403–411, Washington, DC, USA, 2007. IEEE Computer Society.

[4] F. Favarim, J. Fraga, L. C. Lung, and M. Correia. GridTS: A new approach for fault tolerant scheduling in grid computing. In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 187–194, July 2007.

[5] F. Favarim, L. C. Lung, J. da S. Fraga, and M. Correia. Fault-tolerant multiuser computational grids based on tuple spaces. In *International Workshop on Dependability in Service-oriented Grids (WODSOG)*, Oct. 2006.

[6] U. Hönig and W. Schiffmann. Improving the efficiency of functional parallelism by means of hyper-scheduling. In *Proceedings of the 2006 International Conference on Parallel Processing Workshops*, pages 283–290. IEEE Computer Society, 2006.

[7] U. Hönig and W. Schiffmann. A meta-algorithm for scheduling multiple dags in homogeneous system environments. In *Proceedings of the eighteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS '06)*, 2006.

[8] A. Iamnitchi and I. Foster. A problem-specific fault-tolerance mechanism for asynchronous, distributed systems. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, pages 4–13, Washington, DC, USA, 2000. IEEE Computer Society.

[9] J. Leon, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[10] E. Luque, A. Ripoll, P. Hernández, and T. Margalef. Impact of task duplication on static-scheduling performance in multiprocessor systems with variable execution-time tasks. In *ICS '90: Proceedings of the 4th international conference on Supercomputing*, pages 439–446, New York, NY, USA, 1990. ACM.

[11] F. B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984.

[12] J. uk In, P. Avery, R. Cavanaugh, L. Chitnis, M. Kulkarni, and S. Ranka. SPHINX: A fault-tolerant system for scheduling in dynamic grid environments. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 12.2, Washington, DC, USA, 2005. IEEE Computer Society.

[13] J. Weissman and D. Womack. Fault tolerant scheduling in distributed networks. Technical Report CS-96-10, Univ. of Texas at San Antonio, 1996.

[14] G. Wrzesinska, R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Fault-tolerant scheduling of fine-grained tasks in grid environments. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(1), February 2006.