Efficient and Fault-Tolerant Static Scheduling for Grids

Patrick Eitschberger
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
Email: Patrick.Eitschberger@fernuni-hagen.de

Jörg Keller
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
Email: Joerg.Keller@fernuni-hagen.de

***Abstract*—Static task graphs model a variety of parallel applications, and are used to schedule such applications in grid platforms. While the scheduling is static, i.e. done prior to execution, processors might fail or not deliver their performance, especially if the grid comprises nodes with donated time, that may be used or shutdown by their owner at any time. We extend a prior proposal for fault-tolerant grid scheduling with task duplication to also cover situations where tasks take much longer than expected from the schedule as a special kind of fault. Furthermore, we consider the time for communication between dependent tasks when placing duplicates. We evaluate both scenarios with a simulator that injects faults and slowdowns to processors, and workloads from a benchmark suite of task graph with a variety of structures. Our results indicate that the overhead in the fault-free case is negligible, that a processor failure mostly increases the schedule makespan only moderately because duplicates can use gaps in the original schedule, and that the effects of a processor slowdown can partly be mitigated by aborting a (slow) task and running its duplicate.**

***Keywords*-static task scheduling; fault tolerance; grid computing; performance monitoring**

I. Introduction

Task graphs are a well-known model for parallel programs. They comprise a collection of tasks with dependencies, i.e. tasks produce output upon completion, that is needed by a dependent task before it can start. Cyclic dependencies are not allowed. As the dependencies do not fully specify an execution order in a grid with a finite number of processing units, the task graph must be scheduled. Static scheduling converts the task graph prior to execution into a schedule, i.e. for each processing unit, a list of tasks with start and end times is given. The goal of the scheduler typically is to achieve a short makespan, which is the maximum completion time over all tasks. With dynamic scheduling, the assignment of tasks to processing units occurs at runtime. Scheduling is an NP-complete problem, hence schedulers typically use heuristics. While static scheduling normally achieves shorter makespan and does not incur overhead during runtime, dynamic scheduling has the advantage that it can adapt to changes in the computing platform that are unknown prior to execution, at the cost of inferior schedules and runtime scheduling overhead.

An example for a change at runtime is the failure of a processing unit during execution. Another example is a change in performance of a processing unit. Both events frequently occur if a grid is used where computing resources are free because they are donated. The owner of a processing unit might at any moment switch off the unit, which leads to a failure, or might use some of its power for his own computations, so that only a part of its computing power remains available for the grid job. As this fraction may become very small, task runtimes might get very long, so that they appear like a special kind of processor failure. As these events are not too infrequent, grid users have a tension between high performance in the fault-free case (resulting from good schedules and small overhead) and handling of these events in the case of a failure. Thus, a combination of static and dynamic scheduling properties would be desirable.

In a prior work [6], we presented a static scheduling algorithm that handles processor failures with the help of task duplication. This algorithm uses already existing schedules and task-graphs, i.e. concentrates on the placement of duplicate tasks in the schedule. If a processor crashes while executing a task, the schedule still can be completed because the duplicate of that task is always scheduled to a different processor, and thus can complete. The schedule also carefully avoids overhead in the fault-free case by having duplicates only execute if necessary, and if possible in execution gaps between tasks, which frequently exist. Thus, the desired compromise for this kind of event is achieved.

In a first step we include the consideration of the communication costs between the tasks (and duplicates), which is very important in terms of makespan and overhead, but was left out in our previous work for a better readability. In a second step we extend the duplicate placement algorithm to also handle performance slowdown: if the duplicate can complete earlier than the predicted completion of a task, the duplicate is run and the task itself aborted. To our knowledge, this kind of load balancing so far has only been treated by dynamic schedulers. Our scheduling approach avoids centralized decision making at runtime, which might become a bottleneck, but still derives predictions for task completion from the past performance of a processor over an extended period of time.

We evaluate the extended approach with the help of a simulator that inputs a schedule produced by our scheduling approach and randomly injects processor failures or processor slowdowns while executing the schedule. We take the task graphs from a benchmark suite of synthetic task graphs [7] that is organized according to several structural criteria, and thus allows to categorize results for certain types of task graphs. Our results indicate that in about 15% of the task graphs, the execution of a duplicate in case of a slow task (with a slowdown of 50%) leads to a shorter makespan. With respect to fault-tolerance, the extension of the makespan in case of a processor failure is only moderate.

The remainder of this paper is structured as follows. In Section II we briefly review elementary facts and related work about task graph scheduling and fault-tolerant scheduling, respectively. In Section III we present the extended placement algorithm for duplicates, the necessary system support at runtime, and the implementation of placement algorithm and simulator. Section IV presents and analyzes the simulation results. In Section V, we conclude and give an outlook on open problems and future work.

II. Scheduling and Fault-Tolerance

Task scheduling is used for the execution of parallel programs on parallel systems, where several parts (tasks) of such programs are efficiently distributed to the different processing units (PUs) by considering the dependencies between the tasks. The dependencies can be described by a directed acyclic graph (DAG), also known as task graph. In a task graph, the nodes or tasks are attributed with a runtime, and the edges are attributed with a communication cost (or time). A valid schedule which assigns each task a processor, a start time and an end time (where end time is start time plus runtime) must respect the dependencies: if there is an edge from task u to task v, then the start time of v must be at least the end time of u plus the communication cost. If both tasks are mapped to the same processor, the communication cost is typically neglected as the communication bandwidth within a processing unit is much higher than over a network between two processing units. The goal of task scheduling is normally the minimization of the makespan or the maximization of the speedup.

Schedulers might also be classified according to when the schedules are created. In general there are two different methods, the static and the dynamic scheduling. With the static scheduling the schedules are created before they are executed so that no further calculations are needed during the runtime of the programs, which would incur overhead and prolong the makespan. But the consequence is that the schedules cannot be modified afterwards. The dynamic scheduling, where the schedules are created at runtime, is more flexible and can react to unexpected situations at runtime. But in such cases the makespan typically is longer [7][11]. Another option is a hybrid variant, a combination of the static and dynamic scheduling.

In the field of scheduling in distributed systems, fault tolerance is very important. Here the PUs are often at different locations so that an error avoidance is a priori hardly possible. Thus unexpected errors could appear (like transmission errors, or errors that result in failures of one or more PUs). The formation of such errors basically cannot be prevented by a scheduler, but it can be considered with specific fault tolerance aspects during the creation of the schedules.

There are some global software techniques like the transaction principle or replication to handle errors in distributed systems. A special kind of replication is duplication, where the fault tolerance is achieved by copying the data. If the original data crashes, the copy (e.g. the duplicate) can be used to continue the execution.

Fault-tolerant static scheduling algorithms have the goal to achieve a certain level of fault tolerance while increasing the makespan as little as possible. Such an increase in length can be considered in both, the fault free and the fault case. For this we use the measure overhead in percent (in the following only called overhead). The overhead is the deviation from the makespan (MS) with fault tolerance techniques (FTT) and the makespan without fault tolerance techniques and can be modelled by:

$$overhead = \frac{MS_{withFTT} - MS_{withoutFTT}}{MS_{withoutFTT}} \cdot 100 \qquad (1)$$

One example for a fault tolerant scheduling method especially for grids is the so called distributed fault-tolerant Scheduling (DFTS), which is presented in [1]. As target architecture several networks are connected with a wide area network and each network has several PUs and a so called SRM-Unit (Single Resource Manager). With the help of this unit the jobs are split into tasks. After that the tasks are either directly distributed to the PUs, or they are firstly saved and then assigned to earmarked PUs. The fault tolerance is realized with replications of the jobs. For each calculation of a job, the user can set the number of copies to be executed in advance. Then the copies are distributed to the most appropriate SRM-Units and are there executed independent of each other. In this scenario there can already be an overhead in the fault free case, which depends on the usage rate of the PUs. If a random copy has finished, all the other SRM-Units are informed and cancel their corresponding execution. Other fault tolerant scheduling algorithms are presented for example in [5] and [8].

As explained above, the consideration of the communication costs of the tasks that are mapped onto different PUs is also important for task scheduling in distributed systems. For example in [15] a fault-tolerant task scheduling algorithm is presented that especially focuses on reducing the communication costs. In this approach a list-based scheduling algorithm is used. For fault tolerance, multiple copies of the tasks are scheduled onto different processors. To reduce the communication costs, each copy of a task sends its data not to all copies of the successor tasks, but only to one copy. In this way the scheduler can handle mutliple failures with only a moderate increase of the communication costs.

Another aspect which is sometimes considered in task scheduling is to handle or circumvent slowdowns of tasks which result from a high usage rate of a processor. In [10], a hybrid job scheduling mechanism is presented especially for grids, that circumvents such slowdowns by using a backfill-based multi-queue strategy.

While these aspects are only considered seperately in existing algorithms, our scheduling approach combines all mentioned aspects, namely the influence of the communication costs, the fault tolerance by using task duplication and handling slowdowns of tasks as a special kind of failure. Futhermore we also guarantee no overhead in a fault free case, while most existing algorithms only focus on a moderate increase of the makespan in case of a fault.

III. Efficient Fault-Tolerant Scheduling

We start by reviewing the ideas from [6], then improve this approach by consideration of communication times between tasks when placing duplicate tasks into the schedule, and extend it by treating slowly progressing tasks as a special kind of processor failure. We also argue that despite the dynamic elements in our approach, the additional system support and overhead at runtime are small compared to normal static task scheduling. Thus, in comparison to dynamic scheduling, we can achieve a shorter makespan in the fault-free case because we know the complete task graph but can still react to faults dynamically without the overhead introduced by task duplication if used in a straightforward way.

A. *Previous Approach*

Fechner et al. [6] assume to have already an existing schedule for a task graph, which is then extended with duplicate tasks to cover processor failures in a fail-stop model. For each task in the schedule, a duplicate (D) is created and placed statically in the schedule before the execution of the program. The duplicate has to be placed on another PU than the original task, so that the schedule can also be executed in a fault case. In most schedules there are several gaps between the tasks because of the dependencies of the tasks. Thus especially those gaps can be used to place the duplicates in an efficient way. If the gaps are too short, the placement of the duplicates already leads to a shift of all successor tasks and thus leads to an overhead in the fault free case. To avoid this overhead and to minimize the overhead in a fault case, one has to take into account some specialities with the placing of the duplicates.

First of all a placement of the duplicate to start at the same time as the original task seems preferable, so that they could be executed in parallel and thus the overhead in a fault case is as low as possible. But then there would be an increased network usage rate, because the results of both, the duplicate and the original task have to be transmitted to all successor tasks and duplicates. For this reason each duplicate is placed with a small delay, also called slack. Thus a duplicate has to be executed only until the corresponding original task has finished correctly. For that the original task sends a commit command to the PU of the duplicate after its fault free execution. Then the duplicate can abort its execution. So in every situation either only the original task or only the duplicate will finish. Fig. 1 illustrates this context. If a placement of some duplicates is not possible, because of the structure of the gaps, the duplicates are placed on the PUs after the end of the original tasks, and get runtime 0. Such placeholders are then called dummy-duplicates (DD). For the DDs the original runtime is only considered when a fault occurs, as they are not executed in the fault free case. With this technique an overhead in the fault free case can be avoided.

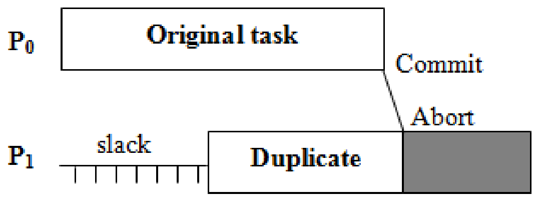

Figure 1. Abort of the duplicate after finishing the original task.

In [6] three different strategies for the placement of the duplicates are presented. In the first strategy only DDs are used. They can be placed into gaps between tasks, or between directly succeeding tasks (considered as a gap of length 0). The schedule remains valid as the DDs are assigned runtime 0 for the fault-free case. Note that a DD can only be placed after the end time of the corresponding original task. The reason for that is that one does not know if the original tasks are correctly executed or if there was a fault before the end of the tasks. With this strategy no overhead in the fault free case can be observed in general, because the DDs are placed with runtime 0. Fig. 2b illustrates an example placement of the DDs in a schedule. The corresponding task graph and input schedule are shown in Fig. 2a. Here the slack and the communication costs are disregarded for a better understanding. This strategy does not take much advantage of the gaps.

The second strategy is an extention of the first strategy. Here the DDs are placed in the same order like before. Then all DDs within the gaps are checked: If there is some idle time before a DD, this DD can be converted into a D. One has to take into account that only such DDs can be converted which are placed at the end time of the corresponding original tasks. The size of the converted Ds is bounded by the length of the idle time before the Ds. In a fault case the execution of the corresponding D has just started, so that only a smaller fraction has to be executed until the end of the D. Thus the second strategy is more efficient in comparison with the first strategy. In Fig. 2c the modified schedule is illustrated.

In the third strategy the gaps in the input schedule are partly extended to fill the gaps more efficiently with Ds. In this strategy we allow a small overhead in a fault free case for a better overhead in a fault case. Fig. 2d illustrates the modified schedule.

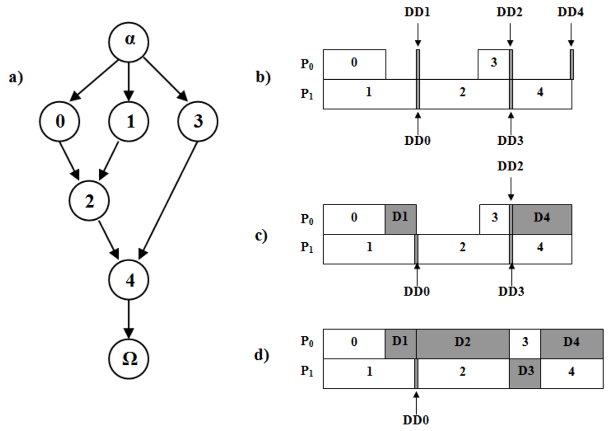

Figure 2. An example task graph and three strategies in comparison.

B. Influence of Communication Time

So far we have neglected the communication cost to transfer the results of one task to the PU of the successor task, if the tasks are on different PUs. The communication cost can have a large influence for the positioning of the tasks and duplicates. This is illustrated by the following small example. We assume that we have two tasks (task1 and task2) and the communication costs between them are for example 5 time units. If the tasks are on the same PU, there is nothing to take into account, because the results of task1 are already on the PU for task2. But if the tasks are on different PUs, the results have to be transferred and thus task2 can start its execution only after the 5 timeunits for the communication. Fig. 3 illustrates the example.

The consideration of the communication costs indicates direct consequences to the three strategies explained above: As a task and its duplicate are always on different PUs, at least one of them is on a PU different from the predecessor task, so that communication cost between succeeding task/duplicate-pairs always occur. Also, as the commit message between task and duplicate incurs some communication cost, a minimum slack is necessary. In the following we demonstrate the consequences exemplary for the first strategy, yet they also apply similarly to the other strategies. Starting from the task graph above, we include an additional task (task5) and an additional edge weight between task1 and task2.

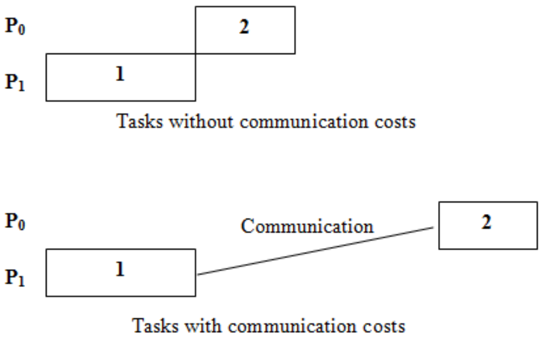
P0
P1
2
1
Tasks without communication costs
P0
P1
Communication
2
1
Tasks with communication costs

Figure 3. The influence of the communication costs.

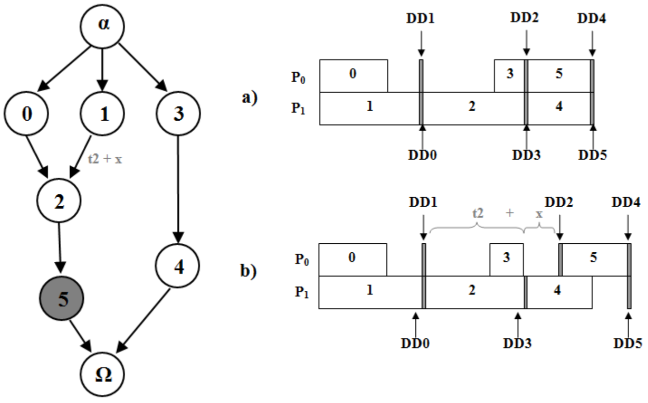

Figure 4. A task graph and schedule a) without and b) with communication costs.

Fig. 4 illustrates the extended task graph and the corresponding schedules without (Fig. 4a) and with communication costs (Fig. 4b). We assume, that the communication costs between task1 and task2 are higher than the computation time of task2 (e.g. with “t2 + x”). We see that the DD2 (and all successors) has to be shifted by x time units and thus the makespan of the schedule is already increased in the fault free case. This example illustrates that one has to take into account all communication costs for all tasks (the original tasks as well as the Ds and DDs). Thus the goal to provide no overhead in the fault free case cannot be achieved with the original strategies. For this reason we devise two further variants for each of the three strategies.

In variant a) the placement of duplicates is executed as in the original version, with the additional consideration of communication times. So in this variant, we tolerate an

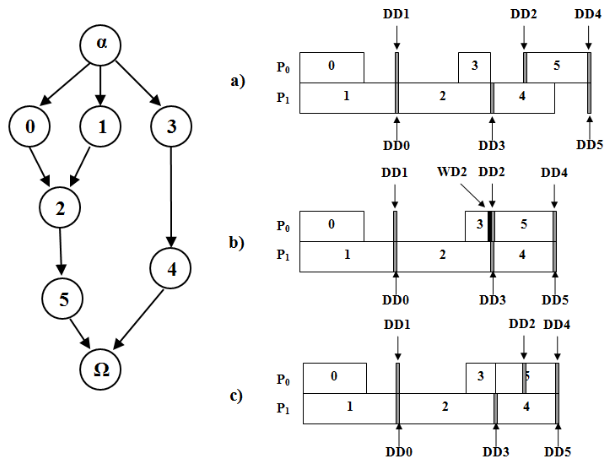

Figure 5. An example task graph and the schedules of the three different variants shown on the first strategy.

overhead in the fault free case (see Fig. 5a).

One first approach to eliminate the overhead in the fault free case (variant b) is to check whether a DD to be inserted leads to a shift of its successor tasks (and thus to an increased overhead) because of the dependencies and communication costs. In this case, if there is a gap between the insert position of the DD and the task before the DD on the same PU, this gap can be used to pull the insert position of the DD forward until either the resulting overhead is undone or there is no more place in the gap. After inserting the DD, a so called wait dummy (see Fig. 5b) is placed before the DD with a computation time of 0 time units in the fault free case, which then in case of a fault extends with the corresponding time of the shifting.

In the last variant c) we consider fault-free and fault case separately. So far we assume that the communication costs always have to be considered. However, if communication costs between DDs and original tasks are not considered (and only in this order), then the schedule has no overhead in the fault free case. If delays result from a DD, they will not be considered initially. Only in a fault case the schedule has to be adapted with the corresponding communication cost and the resulting shiftings. Thus there could be some crossings of original tasks and DDs in the fault free case (see Fig. 5c). If such crossings are allowed no overhead can be guaranteed in the fault free case. Therefore each PU must have the information about the complete task graph and a memory buffer for the results of each task that has just finished. The reason for the buffer is that the transmission of the results will not be aborted in case of a fault and thus the successor tasks on the other PUs can continue their execution. So we get nine versions in total, which are subdivided in three strategies with three variants each.

Note that both the ratio of communication time to computation time (task runtime) and the number (and structure) of dependencies influence both the input schedule and the resulting schedule with duplicates. For example, if the communication times between tasks are small compared to the task runtimes, then our results will not deviate notably from the strategy in [6]. For task graphs with a high ratio of communication to computation, and/or with a high degree of dependency, the input schedule will most likely contain a notable number of gaps. This simplifies the placement of duplicates and should produce a better result.

In the experimental validation, this is taken into account by the structure of the benchmark set, which contains schedules for different classes of task graphs for several criteria (e.g. low/average/high computation to communication ratio, low/average/high degree of dependency). Thus, the effectiveness of the strategies can be compared for the different classes of task graphs.

C. Handling Task Slowdown

So far we described the different kinds of duplicates within the different strategies and variants and explained how they act in a fault case. If there is only a slowdown for a task (instead of a fault), we have to take into account some further rules, because there are only a few cases in which the use of a duplicate could lead to a better makespan and thus to a lower overhead. Fig. 6 illustrates all cases that could lead to a better makespan. In general it could be better to use a duplicate if the end time of the slowed original task is later than the end time of the corresponding D or DD (cases 1 and 2 in Fig. 6). In variant b) also the wait dummies have to be considered (case 3). Important for variant c) (case 4) is that the resulting shift of the DD also must be considered. All other cases like for example a DD which is placed after the end time of the slowed original task are not relevant for an improved overhead.

For handling a failure, a duplicate (D or DD) is started by the PU if the commit message from the original task is not yet received. If that message arrives while the duplicate is already executing, the duplicate is aborted. For handling slowdown, a duplicate should only be started if it terminates faster than the original task. Otherwise, the duplicate would still be aborted by reception of the commit message, but would lead to overhead if the next task on the list of this PU was delayed by the duplicate. Hence, each original task must send a message to its duplicate with an estimate of its actual end time. The message must be sent early enough so that it reaches the PU where the duplicate is placed before the duplicate is to be started. Then the duplicate can compute locally whether one of the cases 1 to 4 is present. In this case, the duplicate sends an abort message to the original task and starts. Otherwise, the duplicate is not started.

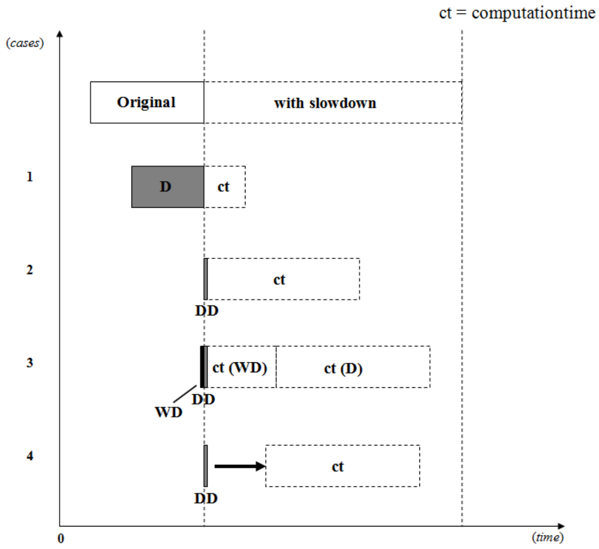

Figure 6. Different cases of placed duplicates compared with a slowed original task.

D. Runtime System Support

Normally, the PUs of the parallel machine executing the tasks only know their own task queue at runtime with some information about the tasks that are mapped onto the PU, like e.g. the task index, the start time, the computation time and the end time of each task in the queue. Because of the dependencies of the tasks a successor task can only start, when it has all results from the predecessortasks. Thus no more information are needed during the runtime to complete the execution of the whole schedule. Yet we see that the dependencies also allow a very simple runtime system: each PU only knows the list of tasks assigned to it, with their start times. The PU launches the next task as soon as the start time has come, and the PU is idle. The task will only start if all its necessary input data is present. Thus, if there is a slowdown of a task, all depending tasks will start later, because they get the results of the slow task later than expected.

Besides starting messages, each PU already in normal static task scheduling must be able to receive messages with input data for tasks assigned to run on that PU in the future. As predecessor tasks might complete much earlier than the successor tasks start, the messages might arrive while a task is run on the PU. To handle processor failures, the PU additionally must be able to receive commit messages and abort the running duplicate, or skip it if it has not been launched yet.

To decide whether the execution of a D or DD could lead to a better makespan in case of a slowdown, the PU of the original task sends a message with the assumed end time based of its usage rate to the PU of the corresponding D or DD, so that this message arrives before the duplicate starts. Then the PU of the D or DD can compare the end time of the duplicate with the assumed end time of the original task. In case of an improvement the PU of the D or DD sends a positive message back to the PU of the original task, which is then aborted. After that the D or DD is started. Otherwise a negative message is sent back and there is nothing else to do. Thus, each PU must be able to launch a message at a time known in advance, and must be able to monitor the performance of a task, i.e. the percentage of CPU time that it gets. We see that those features do not need much extension in the runtime system.

E. Implementation

We have implemented a program that supports all explained strategies and variants. The program can be subdivided into a generator, where the task duplicates are inserted into the existing schedule, and thus the schedule for the fault free case is created, and a simulator, that simulates execution of the schedule, injects failures and slowdowns, and calculates the changes of the schedules in such cases. The structure and workflow of the duplicate placement is presented in Fig. 7.

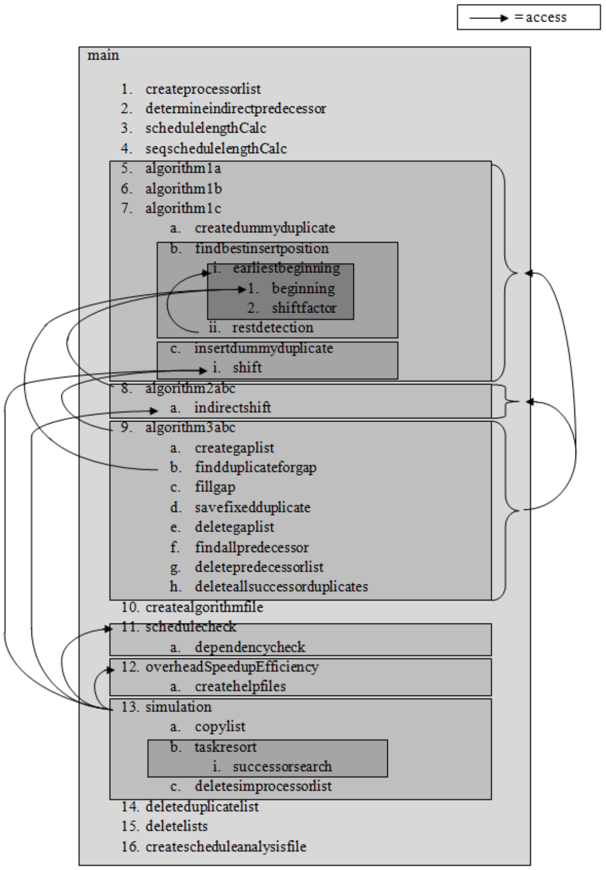

Figure 7. The structure and workflow of the scheduler.

First of all the algorithm uses the information from the existing task graph and schedule, that are given in files, as input to create three schedules (one for each variant). The schedules are represented by singly-linked lists, so called processor lists, where the elements repesent the PUs. Each element (PU) consists of a doubly linked list for the tasks that are mapped onto the corresponding PU. The structure of a task contains among other things information about the task index, the processor index, the start time, the end time, the computation time, the shift factor and the task type. The structure of the taskgraph is held in a vector, where important informations for each task are saved under the corresponding task index. However, there are also some structures like e.g. list of indirect predecessors or successors, that are not necessary for a correct execution but useful for a better performance of the scheduler.

The original schedule is then extended by the first strategy. In variant a) the algorithm creates one DD for each original task and searches for the best insert position. This can be done by finding the earliest insert position in a first step. Therefore the scheduler initially sets the insert position for a DD to the end of the corresponding original task on each PU that differs from the PU of the original task. Then all dependencies and communication costs of the predecessor tasks are considered and the insert position is changed on the PUs where this is necessary. After that the slack is considered (which can be set by the user in the beginning) and the potential shift is calculated by the shifting factor. Finally it must be checked, if the insert position of the DD occurs during the runtime of another task on the corresponding PU. If in such a case the other task is a predecessor of the DD, the insert position is set to the end of this task and the shift factor is corrected if necessary. In the second step the biggest gap around the insert positions is determined with the function “restdetection”. In this function the scheduler first checks the part of the gap that is before the insert position so that the DD could be expanded to a D in the second strategy. Then the remaining part of the gap, which is after the insert position is considered. If this part is too small, there would be a rest of the DD, which in case of a fault would lead to a shift. The best PU to insert the DD is determined with the following decreasing priority: smallest shifting factor, smallest rest, earliest insert position. After finding the best insert position, the DD is mapped on the corresponding PU in consideration of possibly shifts.

The other variants (b and c) have only small modifications of the first variant. In variant b) the scheduler checks the requirements for a wait dummy before the DD is mapped. If it is possible, the insert position and the shift factor are corrected. After that the DD is mapped to the PU, the corresponding shifts are calculated and the schedule is modified. Finally the WD is placed before the DD. In variant c) only the DDs with a shift factor are considered. The shift factors are saved as a property to the corresponding DDs and then they are set to zero. Thus in a fault free case, they are not considered, but in the case of a fault, the task has still the information about the shift.

The other strategies are identical for all variants, because they use the schedules of the different variants from the corresponding previous strategy. Thus, in the second strategy all DDs are searched, which are placed at the time where the corresponding original tasks end. Then these DDs are expanded forward. In the third strategy all gaps in the schedule are searched and saved. Then the best D is determined and inserted in the schedule with an extention of the gap where required. After that all predecessor Ds or DDs are saved and all successor Ds or DDs are deleted. Then the resulting shifts are calculated and the schedule is modified. Finally all strategies are once more executed, so that the third strategy calls itself recursively until there are no more gaps which can be used efficiently.

The simulator does not inject random faults or slowdowns, but systematically applies a failure or slowdown to any task, and computes how the makespan changes. Thus, it is not a discrete event simulator but a quite specialized evaluation program. The simulator distinguishes between failures and slowdowns. The fault-tolerant schedule from the generator is input in a first step. The user can decide at the beginning of the program, if the simulator should inject failures or slowdowns (with a specific percentage rate). Failures are simulated on all PUs and for all mapped original tasks. The simulator starts with the last orignial task of the first PU and sets a failure to the beginning of that task. Then the corresponding D or DD is searched. If it is a DD which has a shift factor, the shift factor is deleted and the tasks are resorted on that PU if necessary. The reason for the resort is that the tasks of a PU are saved in a doubly linked list and thus e.g. crossings of DDs and original tasks cannot be ordered clearly. After that all direct and indircet successor tasks that are not mapped on the PU with the failure are shifted. For a better performance of the scheduler the changed schedule from the simulation before is used to simulate the next earlier failure on the same PU.

Slowdowns are also simulated on all PUs and for all mapped original tasks. In this case the simulator starts with the first original task on the first PU and calculates the computation time of the task with the setting of the percentage rate. Then two cases are simulated. In the first case, the corresponding D or DD is used instead of the slow original task. This is only done if the D or DD complies with the rules explained above that could lead to a better makespan. Thus the original task is aborted when the D or DD is started and the D or DD is then extended. After that all direct and indirect successor tasks are shifted if necessary. In the second case, the schedule is simulated with the slow original task. So in this case the schedule is used like in a fault free case but with some shifts because of the slowdown of the original task. Finally the makespans of both cases are compared and the potential improvement of using the corresponding D or DD is saved.

IV. Experimental Results

We evaluate the scheduling algorithm with the task graphs from the benchmark suite of synthetic task graphs [7], that comprises 36,000 optimal schedules, which differ in the number of PUs (2, 4, 8, 16 and 32), the number of tasks (7 - 12, 13 - 18 and 19 - 24), edge density, the edge length and the node and edge weights. The schedules in this benchmark suite are generated with the Pruned Depth-first Search (PDS) method. In the following the resulting overheads for the whole benchmark suite are presented in the fault free and fault case. Then we analyze the results for the different classes of task graphs. Finally the results for different slowdowns are pesented.

A. Overhead in the fault free and fault case

We extend the schedules with our algorithm and simulate failures on every PU from each task on. Tab. I illustrates the overhead results for all strategies and variants in the fault free case. The overhead decreases from variant a) to c) and increases from strategy 1 to 3. Note that in variant c) there is no overhead in the fault free case for the strategies 1 and 2. Thus we could reach our goal to provide no overhead in the fault free case.

Table I
Overhead in the fault free case.

	Variant a	Variant b	Variant c
Strategy 1	2.52%	1.38%	0.00%
Strategy 2	2.52%	1.38%	0.00%
Strategy 3	3.83%	2.74%	1.39%

In Tab. II the overhead results in the fault case are presented. The values are averaged over all fault positions, i.e. all tasks, and all schedules. With strategy 2c we got the best results for the minimum (4.19%), the average (20.87%) and the maximum (36.83%) overhead. Thus the results of strategy 3, where the gaps in the schedules are increased for a better overhead, are not as good as expected.

Table II
Overhead in the fault case.

	Minimum	Average	Maximum
Strategy 1a	7.43%	24.23%	39.61%
Strategy 2a	6.41%	22.08%	37.18%
Strategy 3a	7.59%	22.76%	37.44%
Strategy 1b	6.45%	24.35%	43.41%
Strategy 2b	5.39%	22.21%	40.84%
Strategy 3b	6.63%	22.85%	40.89%
Strategy 1c	5.28%	22.89%	39.05%
Strategy 2c	4.19%	20.87%	36.83%
Strategy 3c	5.42%	21.54%	37.05%

B. Influence of the different classes of task graphs on the overhead

In the following analysis we concentrate on the influence, that the structure of task graphs has on the overhead, and do not focus on the different strategies and variants. We use the results of strategy 2c for the analysis because this strategy delivered the best results. The identified tendencies and the respective explanations can be transferred to the other strategies. First of all the task graphs can be classified according to the number of PUs (2, 4, 8, 16 and 32). In Tab. III the overhead results are presented.

Table III
Influence of the number of PUs on the overhead.

	Minimum	Average	Maximum
2 PUs	8.86%	37.01%	62.79%
4 PUs	3.27%	17.90%	32.05%
8 PUs	3.00%	16.64%	29.90%
16 PUs	2.88%	16.26%	29.44%
32 PUs	2.82%	16.15%	29.34%

We see that the overhead decreases with an increasing number of PUs. The highest overhead which is more than twice as large as the other values occurs with 2 PUs. In this case also the minimum and maximum is far apart. This results from the small number of choices to map the DDs and Ds onto the PUs. In most cases both PUs are already allocated with original tasks before the mapping of the DDs and Ds. Thus there are only a few possibilities to map the duplicates. There is also the fact that in case of a failure one PU has to execute all existing tasks. This is the worst case, because the tasks can be only executed in a sequential order. In contrast the overhead with 4 PUs is only the half the overhead with 2 PUs. The values with the use of 8, 16 and 32 PUs are nearly the same. In general only a few further PUs are needed to decrease the overhead dramatically. But beyond a certain number, the PUs cannot be utilized fully. Thus some PUs would be unused. This explains the small differences between the overhead values of the higher numbers of PUs (4 to 32).

The task graphs can be also classified according to the number of tasks. The overhead results are presented in Tab. IV.

Table IV
Influence of the number of tasks on the overhead.

	Minimum	Average	Maximum
7-12 tasks	6.61%	25.18%	41.93%
13-18 tasks	3.48%	19.94%	35.99%
19-24 tasks	2.41%	17.38%	32.45%

In general the values show that there is a notably lower overhead with an increasing number of tasks. With the use of many tasks there are basically more gaps and the Ds and DDs can be mapped more efficiently.

The next classification is the edge density. In Tab. V the results are presented.

We can see that the overhead decreases with a higher edge density. The tasks have more dependencies with a higher edge density, so that longer gaps exist between the tasks.

Table V
INFLUENCE OF THE EDGE DENSITY ON THE OVERHEAD.

	Minimum	Average	Maximum
low e.d.	4.20%	24.50%	46.02%
avg e.d.	4.23%	20.72%	35.71%
high e.d.	4.28%	16.75%	27.55%
rand e.d.	4.04%	21.30%	37.60%

Those gaps can be used for the expansion of the duplicates for both the mapping of DDs and Ds and later in case of a failure of one PU. In total there are fewer shiftings and thus a lower overhead in case of a fault.

A further classification is the edge length. The Influence on the overhead is shown in Tab. VI.

Table VI
INFLUENCE OF THE EDGE LENGTH ON THE OVERHEAD

	Minimum	Average	Maximum
EL short	4.23%	18.81%	30.46%
EL avg	4.18%	21.08%	37.23%
EL long	4.00%	22.56%	43.05%
EL rand	4.35%	21.04%	36.55%

The overhead increases with a higher edge length. One reason is that early shiftings directly affect tasks that must be executed much later. Thus in total there would be a high overhead from only a few shiftings. In contrast shiftings, in the case of shorter edge length can usually be compensated by gaps. Furthermore with short edge lengths the shiftings are negligible because of the dependencies to other tasks.

The last classification are the node and edge weights. In Tab. VII the results are presented. The node and edge weights differ between high (H), low (L) and random (R).

Table VII
INFLUENCE OF THE NODE AND EDGE WEIGHT ON THE OVERHEAD

	Minimum	Average	Maximum
HNodeHEdge	4.16%	18.06%	32.57%
HNodeLEdge	2.89%	17.79%	35.57%
LNodeHEdge	5.78%	27.49%	41.28%
LNodeLEdge	4.50%	20.90%	37.23%
RNodeREdge	3.95%	20.66%	37.28%

First of all the input schedules are optimal schedules. The tasks are initially mapped in a way that there are only few transmissions and thus the schedule length is very small. Dependent tasks which would incur long transmission times if mapped on different PUs, are typically mapped onto the same PU. The Ds and DDs in contrast have to be mapped onto other PUs and thus lead to long transmission times. Thus the schedules are notably prolonged. In case of a fault this effect would be enhanced because of the expansion of the Ds and DDs. However, on small tasks with low transmission times the overhead result is similar to the first two cases.

In total the analysis shows that all classes of the task graphs have an influence on the overhead, but there is no class that does not profit from our scheme.

C. Use of duplicates instead of slow tasks

As slowdown we choose the values of 50 %, 70 % and 80 % and test for each slowed original task if there is an improvement by using the corresponding duplicate. As results, we simulate 36,000 schedules for a slowdown of 50 %, 6,800 schedules for a slowdown of 70 % and 10,164 schedules for a slowdown of 80 %, respectively.

In case of 50 % slowdown 510,456 tasks were simulated in total. For 75,846 tasks the use of the duplicate leads to an improvement of the makespan compared to the use of the slowed original tasks. Thus in 14.85 % of all cases the usage of the duplicate results in a better makespan. Tab. VIII presents the results for the different strategies and variants.

For a slowdown of 70 %, 117,909 tasks are simulated. The use of the duplicate leads to an improvement of the makespan for 60,544 tasks, which is 56,28 % of all cases. In Tab. IX the results for the different strategies and variants are presented.

With a slowdown of 80 %, 176,160 tasks are simulated in total. For 125,414 tasks the use of the duplicate leads to an improvement of the makespan. This is 71,19% of all cases. Tab. X presents the results for the different strategies and variants.

Table VIII
IMPROVEMENT OF THE MAKESPAN WITH THE USE OF DUPLICATE IN THE CASE OF 50 % SLOWDOWN

	Minimum	Average	Maximum
Strategy 1a	0.00 %	0.00 %	0.00 %
Strategy 2a	0.98 %	3.24 %	32.13 %
Strategy 3a	0.98 %	3.60 %	32.13 %
Strategy 1b	0.00 %	0.00 %	0.00 %
Strategy 2b	0.98 %	3.74 %	90.09 %
Strategy 3b	0.98 %	4.07 %	90.09 %
Strategy 1c	0.00 %	0.00 %	0.00 %
Strategy 2c	0.98 %	3.03 %	31.25 %
Strategy 3c	0.98 %	3.45 %	31.25 %

Table IX
IMPROVEMENT OF THE MAKESPAN WITH THE USE OF DUPLICATE IN THE CASE OF 70 % SLOWDOWN

	Minimum	Average	Maximum
Strategy 1a	0.13 %	8.01 %	39.36 %
Strategy 2a	0.13 %	8.51 %	39.47 %
Strategy 3a	0.13%	8.49%	39.47%
Strategy 1b	0.13%	8.28%	69.90%
Strategy 2b	0.13%	8.74%	69.90%
Strategy 3b	0.13%	8.73%	68.09%
Strategy 1c	0.13%	8.03%	39.36%
Strategy 2c	0.13%	8.55%	39.36%
Strategy 3c	0.13%	8.55%	39.36%

We observe that up to a slowdown of 50% the DDs in all strategies and variants are not considered because

Table X
Improvement of the makespan with the use of duplicate in the case of 80% slowdown

	Minimum	Average	Maximum
Strategy 1a	0.31%	14.89%	44.43%
Strategy 2a	0.31%	15,34%	44.43
Strategy 3a	0.32%	15.33%	44.43%
Strategy 1b	0.20%	15.10%	84.61%
Strategy 2b	0.28%	15.54%	84.61%
Strategy 3b	0.20%	15.52%	84.69%
Strategy 1c	0.31%	14.93%	43.18%
Strategy 2c	0.31%	15.39%	44.43%
Strategy 3c	0.32%	15.39%	44.43%

the earliest placement of a DD is at the end time of the original task. Thus if the computation time of the original task doubles because of the slowdown, the end time of the slowed orignal task is the same as the end time of the converted DD (without any communication costs). This is the reason why the strategy 1, which uses only DDs, has no duplicates which lead to an improvement of the makespan. Only for a slowdown of over 50% the use of DDs can also lead to an improvement of the makespan (see Tab. IX and Tab. X). The results of the other strategies vary only a little because the sorting of the duplicates differs between the strategies and variants (e.g. different sizes of the duplicates or a different placement). While the average improvement seems small, note that we only slow down a single task whose runtime only comprises a fraction of the runtime, so that large improvements cannot be expected.

V. Conclusions and future work

We have presented a static task scheduling algorithm that takes into account both processor failures and processor slowdowns by using task duplicates, without the need for extensive interaction during runtime. Our results indicate that the runtime overhead is small and that both error cases can partly be improved. As future work, we plan to cover simultaneous slowdown and failure by using two duplicates per task, and to extend our experiments to a prototype system. Furthermore we plan to consider energy efficiency, e.g. to scale the frequency of a PU down for some time (and thus slow down the task) if the slowdown does not increase the makespan, and to trade overhead in the fault case against energy by accelerating duplicates via frequency scaling.

References

[1] Abawajy, J.H.: Fault-Tolerant Scheduling Policy for Grid Computing Systems. In: Proccedings of the 18th IPDPS, 238, IEEE Computer Society, 2004

[2] Adam, T.L., Chandy, K.M., Dickson, J.: A comparison of List Scheduling for Parallel Processing Systems. Communications of the ACM, 17:685-690, 1974

[3] Bansal, S., Kumar, P., Singh, K.: An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. IEEE Transactions on Parallel and Distributed Systems, 14(6), 2003

[4] Braun, T.D. et al.: A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. Journal of Parallel and Distributed Computing. 61(6):810-837, 2001

[5] Favarim, F. et al.: 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), 187-194, 2007

[6] Fechner, B., Hönig, U., Keller, J., Schiffmann, W.: Fault-Tolerant Static Scheduling for Grids. In: Proceedings 13th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'08), 2008

[7] Hönig, U., Schiffmann, W.: A comprehensive Test Bench for the Evaluation of Scheduling Heuristics. In: Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems (PDCS), 2004

[8] In, J. et al.: SPHINX: A Fault-Tolerant System for Scheduling in Dynamic Grid Environments. In: Proceedings of the 19th IPDPS, 12, IEEE Computer Society, 2005

[9] Kasahara, H., Narita, S.: Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. IEEE Transactions on Computers, C-33(11):1023-1029, 1984

[10] Kiejin, P., Changhoon K., Sungsook K.: Hybrid Job Scheduling Mechanism Using a Backfill-based Multi-queue Strategy in Distributed Grid Computing. IJCSNS, VOL.12 No.9, 39-49, 2012

[11] Kwok, Y.-K., Ahmad, I.: Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Surveys, 31(4):406-471, 1999

[12] Kwok, Y.-K.: Parallel Program execution on a Heterogeneous PC Cluster Using Task Duplication. In: Proc. 9th Heterogeneous Computing Workshop, 364-374. IEEE Computer Society, 2000

[13] Papadimitriou, C.H., Yannakakis, M.: Towards an architecture-independent analysis of parallel algorithms. Communications of the ACM, 510-513, 1988

[14] Ritchie, G.: Static Multi-processor Scheduling with Ant Colony Optimization & Local Search. Master's thesis, School of Informatics, University of Edinburgh, 2003

[15] Tabbaa, N., Entezari-Maleki, R., Movaghar, A.: Reduced Communications Fault Tolerant Task Scheduling Algorithm for Mutliprocessor Systems. IWIEE, 3820-3825, 2012.

[16] Tanenbaum, A., van Steen, M.: Distributed Systems — Principles and Paradigms, 2nd ed. Prentice Hall, 2006

[17] Yang, T., Gerasoulis, A.: A fast static scheduling algorithm for DAGs on an unbounded number of processors. In: Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, 633-642. ACM Press, 1991