Anticipated Distributed Task Scheduling for Grid Environments

Thomas Rauber[1], Gudula Rünger[2]

[1] University Bayreuth
Department of Computer Science
Bayreuth, Germany
rauber@uni–bayreuth.de

[2] Chemnitz University of Technology
Department of Computer Science
Chemnitz, Germany
ruenger@informatik.tu–chemnitz.de

Abstract

Heterogeneous distributed environments or grid environments provide large computing resources for the execution of large scientific applications. The effective use of those platforms requires a suitable representation of the application algorithm which makes a distribution of parts of the application across the distributed environment possible. A representation of an application algorithm in form of interacting tasks has been shown to be a suitable programming model for those distributed environments, where tasks can be shipped to remote computing resources for execution. The efficient execution of an application also depends on the time for sending tasks and data to remote resoures, which adds an additional overhead to the distributed execution time. In this paper, we propose a method to overlap the execution of current tasks with the shipping time for tasks to be executed later. The efficient overlapping is achieved by an anticipated scheduling algorithm for the placement of future task executions.

1 Introduction

Scientific applications or numerical algorithms often provide an inherent structure of interacting modules which can be exploited naturally for a parallel or distributed implementation. A suitable parallel programming model for expressing algorithms with a modular structure uses tasks to represent the modules of an application algorithm. Data or control dependencies between tasks of the application form a task graph where nodes represent tasks and arrows between tasks denote dependencies. Tasks may be single-processor tasks or multiprocessor tasks (M-tasks), which can be executed on an arbitrary number of processors, but also on a single processor. M-tasks graphs have two levels of potential parallelism: each M-task can be executed on a set of processors of a parallel platform and different M-tasks can be executed concurrently on different disjoint sets of processors of one platform or on different parallel platforms in a distributed environment. This makes M-task programming ideally suited for distributed or Grid environments combining different parallel platforms.

The execution of M-task programs in distributed heterogeneous environments requires a scheduling of M-tasks of the application program. For an efficient execution, the current structure of a distributed environment as well as the expected (parallel) execution time of M-tasks on different parallel components of the platform is important. The current structure of the distributed computing environment can be described as a varying set of computing components and the dynamic computing load on each of the components. A migration of M-tasks and their corresponding data to other components during runtime of the application program may be required for load balance. Thus, the time for migrating M-tasks and their corresponding data to other platform components has to be taken into account. An approach to efficiently schedule M-tasks in a dynamic distributed client-server environments has been presented in [16]. The appoach considers execution times of M-tasks as well as migration times for M-tasks to remote computing resources. In this paper, we extend this approach to take an overlapping of execution times and migration costs into account.

For M-task scheduling with dynamically changing computation load and/or dynamically changing distributed environments, the scheduling decision has to be done at runtime. Thus, also the migration of tasks has to be done at runtime which may induce a high communication overhead. To reduce the communication overhead we propose an overlapping of computation time and migration time, such that the schedul-

1-4244-0054-6/06/$20.00

ing decision and the migration of tasks is done during the execution of other tasks. An anticipated migration of tasks is useful if the tasks to be executed next are known and are mainly based on data already available, which holds for many applications from scientific computing with modular structure. Those data might be stored on a different platform component and need to be sent to the same location as the task. The efficient realization of an anticipated task placement, however, requires modified scheduling algorithms since the scheduling decision has to be based on previous decisions and has to be able to react to different situations of given task distributions. We propose a scheduling algorithm which starts with a given distribution of current tasks across the computing resources and which decides about the future placement in a distributed way. The distributed decisions cause a small number of migrations since each distributed resource decides on its own computation load and communication requirements. We present the distributed scheduling algorithm based on a layered approach and show suboptimality results.

The rest of the paper is structured as follows. Section 2 introduces the model for programming with multiprocessor tasks and the execution environment. Section 3 presents the scheduling algorithm with anticipated task placement and shows suboptimality bounds. Section 4 discusses related work, and Section 5 concludes the paper.

2 Multiprocessor task programming

In this section, we describe the M-task programming model and the model for the distributed execution environment used for the scheduling.

2.1 Programming model

Multiprocessor task programming is based on a decomposition of an application algorithm into a set of modules which can be realized as M-tasks. Each module realizes a well-defined independent part of the entire application algorithm with input data from other modules and producing output data required by other modules. The input and output data induce dependencies between modules such that output data from a module A are regarded as input data by a module B. Data and control dependencies give rise to a task graph $G = (V, E)$ with the module activations as nodes and edges between nodes representing dependencies. Task graphs due to module program structures can be explicit, i.e. the task graph can be seen in the program structure, or the task graph can be more implicit when iterative or recursive module structures are allowed [10]. A suitable library for M-task programming is Tlib [17].

Each M-task may have an internal structure which arizes from a data-parallel or SPMD computation. But there is no relation between the internal computations of different M-tasks. In the following, we consider an M-task program with input and output dependencies where each M-task can be executed on an arbitrary number of processors of one of the computing resources in a distributed environment. The execution has to guarantee that the input data required by an M-task $M \in V$ is available when the execution of M starts. More precicely, this means:

- The input data must have been produced by predecessor tasks M' of M with $(M', M) \in E$ or are given as input data to the application program.
- The input data of M must be available at the computing resource which executes M. Usually, M expects the input data using a specific data distribution which must either be produced by the corresponding predecessor M' or which must be generated by a specific redistribution operation before the start of M.

Due to the input-output relations, the task graph G of one M-task program forms a directed acyclic graph (DAG). In the context of a Grid environment, a computing resource may have more than one DAG to compute at the same time. The set of DAGs may result from the submission of different M-task programs by different users.

2.2 Distributed environment

The execution model that we assume in the following is a Grid environment where clients specify M-task programs for execution. The environment consists of a set of servers $\{S_1, \ldots, S_n\}$ where each server controls a local execution resource that allows the execution of M-tasks. The local execution resources of a server usually consist of more than one processor and different servers may control execution resources with different numbers of processors. The servers are connected by a grid network such that each server has a direct connection to a set of neighboring servers. A client submits M-tasks programs to its local server for execution. A server obtaining an execution request from its client is responsible for the execution. The execution may be performed completely on the local computing resource, but the execution request may also be forwarded to a neighboring server, if the local server is busy with other

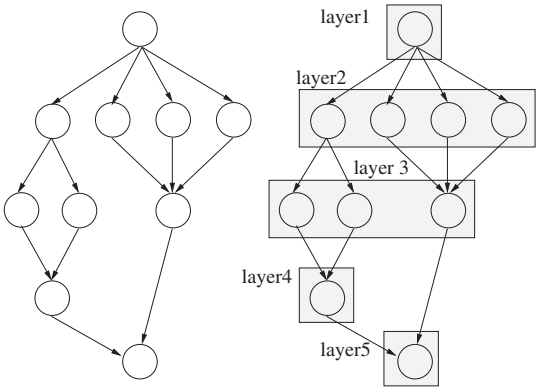

Figure 1. Task graph (left) and partitioning into a sequence of layers (right).

task executions. The execution of an M-task program may be split over the execution resources of several servers.

3 Scheduling approach for dynamic task graphs

For the scheduling of an M-task graph, we use a two-step approach, see [15]. In the first step, the task graph is partitioned into a sequence of layers $L_i, i = 1, 2, \ldots$ such that the tasks of one layer are independent of each other and can therefore be executed concurrently, see Figures 1 for an illustration. The different layers have to be executed sequentially one after another. Usually, there are several possibilities to partition a task graph into layers. The greatest flexibility for the overall scheduling is obtained by using as few layers as possible. This can be achieved by a greedy algorithm that runs over the task graph and puts as many nodes as possible into the current layer. The number of layers is determined by the longest path from the entry node of the task graph to the exit node.

In the second step, the scheduling of the tasks in one layer is determined: The algorithm decides whether the independent tasks in one layer should be executed (concurrently or sequentially) on the local cluster or whether tasks should be forwarded to other servers for execution. To make this decision, the following issues have to be taken into consideration: (1) the local resources available to the server, their current workload and the number of M-tasks waiting for execution; (2) the workload and resources available on neighboring servers; (3) the expected execution time and migration costs of the M-tasks of the current layer, mainly determined by the costs of moving the data required to another server. In [16], we have presented an algorithm to make the decision which of the tasks of a server should be forwarded to another server and which of the tasks should be computed locally. In this article, we extend the algorithm presented in [16] such that the decision of the task placement of one of the layers is taken before the computation of the previous layer. The goal is to hide the necessary communication times for task migration as far as possible behind the computations for the previous layer. We also give suboptimality bounds for the new algorithm.

3.1 Scheduling with anticipated task placement

The decision for the placement of the tasks of layer L_{i+1} is taken after the task placement of layer L_i, $i = 1, 2, \ldots$. Before starting the execution of the tasks in L_i, the placement of the tasks of layer L_{i+1} is determined. Thus, the necessary task migration can be performed during the task execution of layer L_i for those tasks whose input data is already available. The goal is to hide the migration costs of the tasks of the next layer L_{i+1} as far as possible. For the placement of the tasks of L_{i+1}, we assume that the *total execution costs* $T_i(S)$ of the tasks of layer L_i is known for each of the servers S of the execution environment.

Based on $T_i(S)$, a task M of layer L_{i+1} at server S should only be migrated, if the migration costs $C(M)$ of M can be hidden by the execution time of the tasks of layer L_i for server S, i.e., if $C(M) < T_i(S)$. According to this observation, we partition the tasks of server S of layer L_{i+1} into migratable and not-migratable tasks as follows:

- The set of not-migratable tasks $\mathcal{M}^{i+1}_{nmig} = \{M'_1, \ldots, M'_l\}$ of server S contains all tasks whose input data is not available yet and those tasks whose expected migration costs exceed the execution time of layer L_i on server S, i.e., those M'_j, $j = 1, \ldots, l$, for which we have:

$$T_i(S) < C(M'_j). \tag{1}$$

  The migration costs of the not-migratable tasks cannot be hidden by the execution of the tasks of layer L_i at server S. Hence, it is not beneficial to migrate a not-migratable task of layer L_{i+1} to a neighboring server, since the execution of such a task can be started earlier on this server than it arrives at other servers.

- The set of migratable tasks $\mathcal{M}^{i+1}_{mig} = \{M_1, \ldots, M_m\}$ of server S contains all tasks

whose migration costs can be hidden by the execution time of layer L_i on server S, i.e. for all tasks M_j, $j = 1, \ldots, m$, we have:

$$T_i(S) \geq C(M_j). \tag{2}$$

The migratable and not-migratable tasks are determined for each of the servers of the execution environment.

In the following, $T_{\text{nmig}}(S, L_{i+1})$ denotes the expected accumulated execution time of the not-migratable tasks of server S for layer L_{i+1} and $T_{\text{mig}}(S, L_{i+1})$ is the expected accumulated execution time of the migratable tasks of server S for layer L_{i+1}. We consider a server S with neighboring servers $S_1, \ldots, S_{n_S}$. S sends migratable tasks to its neighboring servers as long as there exists a server S_j, $j = 1, \ldots, n_S$ for which

$$T_{i+1}(S_j) < T_{i+1}(S) \tag{3}$$

with $T_{i+1}(S_j) = T_{\text{nmig}}(S_j, L_{i+1}) + T_{\text{mig}}(S_j, L_{i+1})$, i.e., $T_{i+1}(S_j)$ is the total accumulated task execution time of layer L_{i+1} on server S_j. If there are several neighboring servers for which Equation (3) is fulfilled, the server S_j with the smallest value of $T_{i+1}(S_j)$ is selected. If there is such a neighboring server S_j, S selects a migratable task M for which the following conditions are fulfilled:

1. After the migration of M to S_j, Equation (3) is still fulfilled.
2. Task $M \in \mathcal{M}_{mig}^{i+1}$ is selected such that the execution time of M is as large as possible.

To find suitable tasks for migration, the migratable tasks of layer L_{i+1} of S are sorted according to decreasing values of $T(M, p_{\max})$ where $T(M, p_{\max})$ describes the expected execution time of task M on S, assuming that S controls $p_{\max}$ homogeneous execution units. Hence, the migratable task with the largest value of $T(M, p_{\max})$ is migrated first to server S_j. If task M is migrated, then this migration changes the value of $T_{\text{mig}}(S_j, L_{i+1})$ or $T_{\text{nmig}}(S_j, L_{i+1})$ for the target server S_j, depending on whether M is a migratable or not-migratable task for server S_j.

The selection for migration is repeated until no target server is available for which Equation (3) is fulfilled. The placement algorithm avoids the circular placement of tasks, since tasks are only migrated from servers with larger values of $T_{\text{nmig}}(S, L_{i+1}) + T_{\text{mig}}(S, L_{i+1})$ to servers with smaller values of this expression and since tasks are only migrated as long as Equation (3) is fulfilled after the migration. The scheduling algorithm for the anticipated tasks placement of one layer is given in Figure 2.

The placement of the tasks of layer L_{i+1} is based on the situation of the execution environment before the execution of the tasks of layer L_i. During the execution of L_i, the execution environment may change because new servers are added to the environment or because existing servers are removed from the system. This dynamic change of the execution environment may make the scheduling decision sub-optimal. But the execution environment is usually not changed very often. Moreover, the placement of the tasks is chosen such that it is completely hidden by the execution time of the tasks of layer L_i, i.e., the task placement essentially comes for free because of the overlapping of migration and computation costs.

3.2 Suboptimality Bounds

In the following, we analyze the schedules produced by the distributed scheduling algorithm in Figure 2. We first consider task graphs with single-processor tasks and then generalize the result to task graphs with M-tasks in the next subsection.

For single-processor tasks, the following suboptimality property holds:

Lemma 1 *Let $S_1, \ldots, S_n$ be servers of the same speed connected by a complete graph and let each server control a single processor. If all tasks of layer L_{i+1} are single-processor tasks and if all tasks of layer L_{i+1} are migratable, then for the total execution time T_{i+1} computed by the anticipated scheduling algorithm the following holds*

$$T_{i+1} \leq T_{opt} + 2 \cdot T(M_x) \tag{4}$$

where T_{opt} ist the execution time of an optimal schedule of layer L_{i+1} and $T(M_x)$ is the execution time of the smallest task M_x of the server with the largest accumulated execution time.

Proof: We sort the servers $S_1, \ldots, S_n$ according to decreasing accumulated execution time for layer L_{i+1} of a task graph. After the algorithm has stopped, no task can be migrated from S_1 to one of the other servers. This is particularly true for the M-task M_x of S_1 with the smallest execution time.

Let $T_{i+1}(S_n)$ denote the accumulated execution time of S_n. Since there are no waiting times for any of the servers, an optimal schedule at least needs execution time $T_{i+1}(S_n)$, i.e., it is $T_{opt} \geq T_{i+1}(S_n)$.
Since M_x has not been migrated from S_1 to S_n, the

```
for each server S {
    sort the neighboring servers S_1, ..., S_nS of S according to increasing values of T_{i+1}(S_j);
    sort the migratable tasks M_mig^{i+1} = {M_1, ..., M_m} of layer L_{i+1} of S according to
      decreasing values of T(M, p_max);
    while (there exists a server S_j with T_{i+1}(S_j) < T_{i+1}(S)) {
      select the first task M from M_mig^{i+1};
      if (T_{i+1}(S_j) + T(M, p'_max) < T_{i+1}(S) - T(M, p_max)) {
        migrate task M to server S_j;
        re-compute execution times of S and S_j and re-sort S_1, ..., S_nS;
      }
      else {
        add M to set of not-migratable tasks of S;
        T_nmig(S, L_{i+1}) = T_nmig(S, L_{i+1}) + T(M, p_max);
      }
}}
```

Figure 2. Anticipated scheduling of migratable M-tasks $M_1, \ldots, M_m$ of one layer for server S.

following inequality holds according to the scheduling algorithm:

$$T_{i+1}(S_1) - T(M_x) \quad \leq T_{i+1}(S_n) + T(M_x)$$

Thus, we get

$$\begin{aligned} T_{i+1}(S_1) &\leq T_{i+1}(S_n) + 2 \cdot T(M_x) \\ &\leq T_{opt} + 2 \cdot T(M_x) \end{aligned}$$

Since S_1 has the largest accumulated execution time, we have $T_{i+1}(S_1) = T_{i+1}$ which leads to Inequality (4). □

The suboptimality bound (4) of Lemma 1 can be expressed in a different way by using the ratio α between the execution time of the task M_x with the smallest execution time and the execution time of all other tasks of server S_1, i.e.

$$T(M_x) = \alpha \cdot (T_{i+1} - T(M_x)) \tag{5}$$

This is formulated in the following corollary.

Corollary 1 *Given the assumptions of Lemma 1 and the assumption $0 < \alpha < 1$ for α defined in (5), then the accumulated execution time T_{i+1} of the anticipated scheduling algorithm fulfills the suboptimality bound:*

$$T_{i+1} \leq \frac{1+\alpha}{1-\alpha} T_{opt}. \tag{6}$$

Proof: It is $T_{i+1}(S_1) = T_{i+1}$ according to the sorting of the servers. ¿From Equation (5), we obtain $T(M_x) = \frac{\alpha}{\alpha+1} T_{i+1}$. Substituting $T(M_x)$ in Equation (4) yields

$$T_{i+1} \leq T_{opt} + 2 \cdot T(M_x) = T_{opt} + 2 \cdot \frac{\alpha}{\alpha+1} T_{i+1}$$

Hence, we get

$$T_{i+1}(1 - \frac{2\alpha}{\alpha+1}) \leq T_{opt}$$

which results in

$$T_{i+1}(S_1) \leq \frac{1+\alpha}{1-\alpha} T_{opt}$$

□

For large numbers of tasks, we usually have values of α, $0 < \alpha < 1$, near 0. Since $(1+\alpha)/(1-\alpha) \to 1$ for $\alpha \to 0$, the suboptimalty bound of the anticipated scheduling algorithm converges to 1, i.e. is close to 1 for small α.

3.3 Suboptimality bounds for M-task graphs

In the following, we generalize Lemma 1 and Corollary 1 to M-tasks. We assume that Server S_i controls p_i processors and that all processors in the system are identical. Furthermore, we assume that each task has linear speedup, i.e. for servers S_j and S_l, $j \neq l$, and for each M- task M, we have

$$p_j \cdot T(M, p_j) = p_l \cdot T(M, p_l) \tag{7}$$

where S_j controls p_j execution units and $T(M, p_j)$ is the execution time of M-task M on p_j processors, $j = 1, \ldots, n$. We again sort the server according to decreasing accumulated execution times. Let $S_1, \ldots, S_n$ be the resulting order. We first show that the accumulated execution time $T_{i+1}(S_n)$ of the server S_n with the smallest accumulated execution time is a lower bound for the execution time T_{opt} of an optimal schedule.

Proposition 1 *For M-tasks with linear speedup, it is $T_{i+1}(S_n) \leq T_{opt}$, where T_{opt} is the execution time of an optimal schedule.*

Proof: We first show that

$$T_{opt} = \frac{1}{\sum_{j=1}^{n} p_j} \sum_{j=1}^{n} p_j \cdot T_{i+1}(S_j) \tag{8}$$

holds. To show this, we assume that

$$T_{opt} < \frac{1}{\sum_{j=1}^{n} p_j} T^*$$

where $T^* = \sum_{j=1}^{n} p_j \cdot T_{i+1}(S_j)$ is the total work needed for all tasks. This inequality implies that there is a schedule with a smaller total work than T^*. Hence, there exists at least one M-task M on at least one server S_j which can be executed in a smaller amount of time on another server S_l, i.e.

$$T(M, p_j) \cdot p_j > T(M, p_l) \cdot p_l$$

But because M has linear speedup, we have on the other hand

$$T(M, p_j) \cdot p_j = T(M, p_l) \cdot p_l.$$

and thus, Equation (8) is true.

We now show that

$$T_{i+1}(S_n) \leq \frac{1}{\sum_{j=1}^{n} p_j} \sum_{j=1}^{n} p_j \cdot T_{i+1}(S_j) \tag{9}$$

To show this, we assume that

$$T_{i+1}(S_n) > \frac{1}{\sum_{j=1}^{n} p_j} \sum_{j=1}^{n} p_j \cdot T_{i+1}(S_j)$$

¿From this equation, we get

$$\sum_{j=1}^{n} p_j \cdot T_{i+1}(S_j) < \sum_{j=1}^{n} p_j \cdot T_{i+1}(S_n)$$

and

$$\sum_{j=1}^{n} [p_j (T_{i+1}(S_j) - T_{i+1}(S_n))] < 0$$

But since $p_j > 0$ and since $T_{i+1}(S_j) > T_{i+1}(S_n)$ because S_n is the server with the smallest accumulated execution time, the last inequality cannot be true. This proves Equation (9) and the claim of Proposition 1. □

Using Proposition 1, we can show the following

Lemma 2 *Let $S_1, \ldots, S_n$ be servers connected by a complete graph and let server S_j control p_j identical processors. Furthermore let the processors of different servers be identical. If all tasks are M-tasks with linear speedup according to Equation (7) and if all tasks of level L_{i+1} are migratable, then for the total execution time T_{i+1} of the schedule of level L_{i+1} computed by the anticipated scheduling algorithm, the following holds:*

$$T_{i+1} \leq T_{opt} + \left(1 + \frac{p_1}{p_n}\right) T(M_x, p_1) \tag{10}$$

where T_{opt} is the execution time of an optimal schedule and where $T(M_x, p_1)$ is the execution time of the smallest task M_x on the server with the largest accumulated execution time.

Proof: We sort the servers $S_1, \ldots, S_n$ according to decreasing accumulated execution time for layer L_{i+1} of a task graph. After the anticipated scheduling algorithm has stopped, M_x cannot be migrated from S_1 to S_n. Therefore

$$\begin{aligned} T_{i+1}(S_1) - T(M_x, p_1) &\leq T_{i+1}(S_n) + T(M_x, p_n) \\ &= T_{i+1}(S_n) + \frac{p_1}{p_n} T(M_x, p_1) \end{aligned}$$

because of Equation (7). Since $T_{i+1}(S_1) = T_{i+1}$ and $T_{i+1}(S_n) \leq T_{opt}$ because of Proposition 1, we get

$$\begin{aligned} T_{i+1} = T_{i+1}(S_1) &\leq T_{i+1}(S_n) + (1 + \frac{p_1}{p_n}) T(M_x, p_1) \\ &\leq T_{opt} + (1 + \frac{p_1}{p_n}) T(M_x, p_1) \end{aligned}$$

which proofs the lemma.
□

Similarly to Corollary 1, we consider a ratio α defined by

$$T(M_x, p_1) = \alpha \cdot (T_{i+1}(S_1) - T(M_x, p_1)) \tag{11}$$

where M_x is the task with the smallest execution time. We use α to express the suboptimality bound given in the next corollary.

Corollary 2 *Given the assumptions of Lemma 2 and assuming $0 \leq \alpha < 1$ and $1 - \alpha \cdot \frac{p_1}{p_n} > 0$ for α defined in Equation (11), the accumulated execution time T_{i+1} of the anticipated scheduling algorithm has the following suboptimality bound:*

$$T_{i+1} \leq \frac{1 + \alpha}{1 - \alpha \cdot \frac{p_1}{p_n}} T_{opt} \tag{12}$$

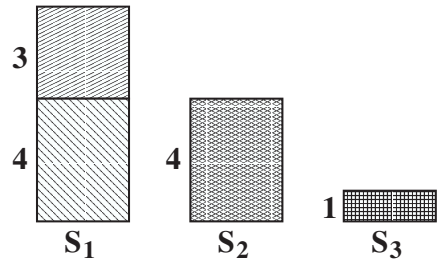

Figure 3. If the servers S_1, S_2, S_3 are connected by a linear array, then the task with 3 work units will not be migrated to S_3 by the distributed scheduling algorithms, although this would lead to a smaller accumulated execution time.

Proof: From the definition of α in Equation (11), we obtain

$$T(M_x, p_1) = \frac{\alpha}{1+\alpha} T_{i+1}(S_1)$$

Substituting $T(M_x, p_1)$ into Equation (10) yields

$$\begin{aligned} T_{i+1}(S_1) &\leq T_{opt} + \left(1 + \frac{p_1}{p_n}\right) T(M_x, p_1) \\ &= T_{opt} + \left(1 + \frac{p_1}{p_n}\right) \frac{\alpha}{1+\alpha} T_{i+1}(S_1) \end{aligned}$$

Therefore, we get

$$T_{i+1} = T_{i+1}(S) \leq \frac{1+\alpha}{1 - \alpha \cdot \frac{p_1}{p_n}} \cdot T_{opt}$$

since $1 - \alpha \cdot \frac{p_1}{p_n} > 0$ □

Again, for a large number of tasks, we have α near 0 and therefore $\frac{1+\alpha}{1-\alpha \cdot \frac{p_1}{p_n}} \to 1$, i.e. the algorithm generates schedules which are nearly optimal.

If the servers are not connected by a complete graph, the structure of the server interconnection may inhibit an exchange of tasks when there is no direct interconnection between servers of different accumulated execution times, see Figure 3 for an example. This may lead to a total accumulated execution time which is larger than for a complete graph.

4 Related work

The scheduling of computational resources is an important issue for Grid platforms and many execution environments provide a scheduling component. Examples of such systems are I-WAY [5], Condor [11], Condor-G [6], Legion [3] and Prospero [13]. But most of these systems aim at the scheduling of complete, independent, coarse-grained jobs that are submitted by independent users. In contrast, the approach presented in this article considers the scheduling of tasks belonging the same application, but takes into consideration that there may be different applications to be scheduled at the same time.

In addition to these application-independent approaches, there are several Grid scheduling techniques that have been developed for a specific application or a specific class of applications, and it is not necessarily straightforward to extend them to other applications or classes [18, 20]. A general approach for arbitrary applications has been proposed in [4]. The approach is based on a decoupling of the scheduling decision from the application-specific information by encapsulation of the application characteristics in an analytical performance model and a data mapper. The core of the scheduler is a general-purpose search procedure identifying useful schedules that can then be evaluated with the performance model. A related approach has been developed in the context of the Application-Level Scheduling Project (AppLeS), see [2] for an overview.

A comparison of different grid scheduling methods for independent coarse-grained tasks has been given in [7], including DFPLTF (dynamic fastest processor to largest task first) [14], Suffrage-C, Min-min and Max-min [12], WQ (work queue) [8], as well as a new algorithm based on a ring organization of the tasks. A min-min heuristics for grid task scheduling has also been presented in [9]. The use of performance prediction techniques to obtain a priori estimates of task execution times which can then be used for the scheduling decision is considered in [19]. That paper presents the performance prediction system PACE and shows how it can be used in combination with a genetic algorithm to select schedules for independent tasks.

Grid scheduling techniques in the context of the Grid Application Development Software (GrADS) project have been presented in [1]. This approach uses a scheduling of workflow graphs which is based on a ranking of the computing resources, reflecting the fit between the component to be executed and the resources available for execution. For the actual scheduling, one of three pre-defined heuristics is selected.

5 Conclusions

In this paper, we have presented a new distributed algorithm for the dynamic scheduling of multiprocessor tasks with dependencies in a Grid environment. The new algorithm tries to hide the migration costs of tasks by deciding on the migration of tasks before the execu-

tion of their predecessor tasks has been started. Thus, the migration can be performed during the execution of the predecessor tasks. A detailed analysis of the algorithm shows that the resulting schedules exhibit a suboptimality bound near 1, if a large number of parallelizable tasks is scheduled and if we assume that the structure of the interconnection network does not inhibit the exchange of M-tasks if this is beneficial. Based on these assumptions, the schedules generated by the algorithm can be proven to be efficient.

The algorithm can be extended in several ways. The current version of the algorithm considers the execution and migration of two consecutive layers of the task graph. This could be extended to more than two layers: If the execution time of the tasks of a layer allows the hiding of the migration costs of more than one of the subsequent layers, the anticipated migration could be extended to these additional layers, provided that the tasks of this layers are already known. The current version of the algorithm migrates a task only if the necessary migration costs can be completely hidden by the execution time of the tasks in the preceding layer. This could be generalized to allow a migration also if this is not the case.

References

[1] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, L. B., X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan. New Grid Scheduling and Rescheduling Methods in the GrADS Project. *International Journal of Parallel Programming*, 33(2):209–229, 2005.

[2] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the Grid using AppLeS. *IEEE Trans. Parallel Distrib. Systems*, 14(4):369–382, 2003.

[3] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource Management in Legion. In *Proc. of the 5th workshop on Job scheduling strategies for parallel processing*, 1999.

[4] H. Dail, F. Berman, and H. Casanova. A decoupled scheduling approach for Grid application development environments. *J. Parallel Distrib. Comput.*, 63:505–524, 2003.

[5] I. Foster, J. Geisler, B. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-Way high-performance distributed computing experiment. *Concurrence: Practice and Experience*, 10(7):567–581, 1998.

[6] J. Frey, T. Tannenbaum, and et al. A Computation Management Agent for Multi-Institutional Grids. *J. Cluster Comput.*, 5:237–246, 2002.

[7] N. Fujimoto and K. Hagihara. A Comparison among Grid Scheduling Algorithms for Independent Coarse-Grained Tasks. In *SAINT 2004 Workshop on High Performance Grid Computing and Networking*, pages 674–680. IEEE, 2004.

[8] R. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[9] X. He, X. Sun, and G. von Laszewski. QoS guided min-min heuristic for grid task scheduling. *J. Computer Science and Technology*, 18(4):442–451, July 2003.

[10] S. Hunold, T. Rauber, and G. Rünger. Multilevel Hierarchical Matrix Multiplication on Clusters. In *Proc. of the 18th Annual ACM International Conference on Supercomputing, ICS'04*, pages 136–145, June 2004.

[11] M. Litzkow, M. Livry, and M. Mutka. Condor – A Hunter of Idle Workstations. In *Proc. of the 8th International Conference of Distributed Computing Systems*, pages 204–211, 1988.

[12] M. Maheswaran, S. Ali, H. Siegel, D. Hensgen, and R. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proc. of the 8th IEEE Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, 1999.

[13] B. C. Neumann and S. Rao. The Prospero resource manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 1994.

[14] D. Paranhos, W. Cirne, and F. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Proc. of the International Conference on Parallel and Distributed Computing (Euro-Par)*, LNCS 2790, pages 169–180, 2003.

[15] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.

[16] T. Rauber and G. Rünger. M-Task-Programming for Heterogeneous Systems and Grid Environments. In *Proc. of the IPDPS'05 workshop on High-Performance Grid Computing, CD-ROM*, 2005.

[17] T. Rauber and G. Rünger. Tlib - A Library to Support Programming with Hierarchical Multi-Processor Tasks. *J. Parallel Distrib. Comput.*, 65(3):347–360, 2005.

[18] S. Smallen, H. Casanova, and F. Berman. Applying scheduling and tuning to on-line parallel tomography. In *Proc. of Supercomputing 2001*, Denver, Colorado, USA, 2001.

[19] S. Spooner, S. Jarvis, J. Cao, S. Saini, and G. Nudd. Local Grid Scheduling Techniques Using Performance Prediction. In *IEE Proceedings - Computers and Digital Techniques*, volume 150(2), pages 87–96. IEE, 2003.

[20] J. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *J. Cluster Comput.*, 1(1):109–118, 1998.