An Enhanced Application Model for Scheduling in Grid Environments

Christoph Ruffner, Pedro José Marrón, Kurt Rothermel
University of Stuttgart
Institute of Parallel and Distributed Systems (IPVS)
Universitaetsstr. 38
D-70569 Stuttgart, Germany
{Ruffner, Marron, Rothermel}@informatik.uni-stuttgart.de

Abstract

Application scheduling in Grid computing requires information about tasks in each application, as well as a description of the available resources that the scheduler needs, to find an efficient assignment of tasks to resources. In this context, information about applications is provided by the application model. Most current models are restricted to only a few features, but in some cases, such as simulations, which are an important example of Grid applications, more detailed information is available. In this paper, we propose an enhanced application model based on the concept of task refinements, which provide the scheduler with fine-grained information and allow it to determine more efficient schedules than traditional approaches. We backup and show the feasibility of our model by means of experimental evaluations.

1. Introduction

One of the main goals of Grid Computing is to provide standardized access to a pool of heterogeneous resources that can be spread around the world [9]. Additionally, resources are shared among authorized users who want to run their applications in the Grid.

Resource management in computational Grids provides the ability to determine whether or not resources are available, and if so, map the submitted applications to specific resources [3]. Therefore, a resource manager needs to be able to perform the following services in an efficient way: authentication services, information services (discovery), deployment, monitoring, and scheduling services. In this paper, we focus on the efficient implementation of the scheduling service.

More specifically, the functionality of a scheduler focuses on deriving schedules for applications by mapping their tasks, i.e. the executable units, to suitable and available resources [16]. In the general case, the scheduling problem is known to be NP-complete, so that the use of heuristics is necessary for its efficient implementation. Depending on the heuristic, additional, more detailed information has to be provided from the application (or the application model) to the scheduler.

Currently used application models [10][11][17] describe the tasks of an application as a block of instructions to be executed on resources. The structure of a task is not further refined, and so, no information is given about the structure and/or the behavior of the application. Regarding communication between tasks, only the models defined in [10][17] make reference to it, although given the system model where Grid computing is applied, network operations play a crucial role [14], especially since the resources are usually located on a heterogeneous network.

As we know from the modeling of simulation applications [2][15], communication can be defined in more detail, so as to provide a sufficient level of granularity to allow the scheduler to make better decisions. In these models, the concept of alternating computation and communication blocks within an application provides a more detailed structure that can be used by the scheduler to generate a more efficient resource allocation.

In this paper we show that this particular type of modeling, used in combination with further refinements for the description of the tasks involved in an application, results in a more efficient schedule. If there are cases where this information is not available, our model adapts to the amount of information available and is able to provide as good a schedule as the simpler algorithms described above.

The paper is structured as follows: Our refined application model for scheduling in Grid environments is presented in section 2. Section 3 defines the reservation model as required by our enhanced application model and section 4 provides details on the calculation of the execution time for a given application. Section 5 introduces a new scheduling heuristic for the application model, which is evaluated in section 6. Section 7 describes related work in the area of application modeling. Finally, section 8 concludes this paper.

2. Application Model

Our application model consists of a set of components organized in tasks that perform computations and a set of relations defined between these components.

2.1 Tasks and components

A *task* t_i defines an executable unit to be scheduled onto one resource. It consists of at least one component k_1 that contains information about the number of instructions, denoted by $|k_1|$, that have to be executed to perform task t_i correctly (Figure 1a). If the task does not communicate with any other task, its execution time is the number of its instructions divided by the speed of the resource in millions of instructions per second (MIPS).

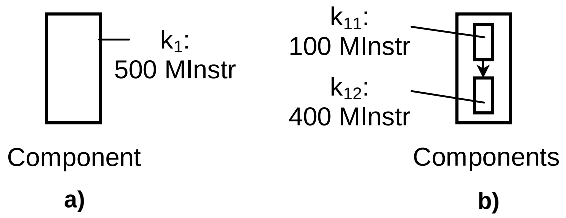

Figure 1: Components of tasks

In contrast to other application models, we introduce the concept of *components* that refine the structure of a task t_i. Therefore, t_i can be divided into components $\{k_1, \ldots, k_n\}$ arranged in a sequential total order "$\rightarrow$". Each component k_j has information about the number of instructions $|k_j|$ to be performed, and so, the total number of instructions of a task can be computed by summing up the number of instructions of its components. Moreover, the order "$\rightarrow$" in which components are connected with each other defines the execution order of the instructions executed on a given resource. In Figure 1b, the order $k_{11}\rightarrow k_{12}$, shown by the arrow, defines that the first instruction of k_{12} is executed after the last instruction of k_{11}.

A component k_j can contain further subcomponents $\{k_{j1}, \ldots, k_{jm}\}$, also arranged in a sequential total order: $k_{j1}\rightarrow k_{j2}$, $k_{j2}\rightarrow k_{j3}$, …, $k_{jm-1}\rightarrow k_{jm}$. The sequential total order implies that if instructions of k_j are executed before instructions of a component k_i, where k_i is refined by $\{k_{i1}, \ldots, k_{in}\}$, then $k_{jm}\rightarrow k_{i1}$ must hold, because the first instruction of k_i is executed after the last instruction of k_j. Therefore, we do not consider intra task parallelism in this paper.

Using the concept of components, a task t_i can be defined by a hierarchy of components, where the *leaves* of the hierarchy define the number of instructions to be executed and the order "$\rightarrow$" between components defines the execution order of their instructions.

The hierarchical model provides the capability to include as much information about the structure of a task as the user knows. If there is no information about the structure, only the instructions are defined for the given task. As we will show in the evaluation later on, the refinement of tasks in this way improves the scheduling of the application.

2.2 Communication Relations

A communication relation is defined between two components of different tasks, if both components communicate with each other during execution of their instructions. This relation needs the following information (Figure 2):

- Communication direction
- Start time of data transfer for each component
- Type of communication
- Amount of data transmitted

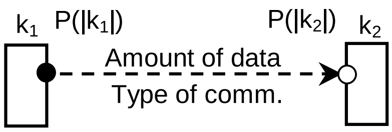

Figure 2: Communication relation

In Figure 2, component k_1 sends data to component k_2 as shown by the *direction* of the arrow.

The *start time* defines the point in time when both components start communicating. The exact point in time cannot be determined in general for an application, since the execution path is not known in advance. Therefore, we define a function $P(|k_i|) = a_i * |k_i|$ where $a_i \in [0..1]$, which defines the start time where component k_i potentially communicates. We assume that communication can take place between the point of time defined by function P and the end of the component, so that a time frame for communication with a given uncertainty is defined. As shown in Figure 2, the function P has to be defined for both components involved in the communication.

As described for different communication platforms such as MPI [12] or PVM [7], the *type of communication* in our model can be asynchronous or synchronous. For our purposes, synchronous communication assumes send and receive operations to be blocking. Asynchronous communication is defined by a non-blocking send operation, where the data to be sent is buffered and can be read by the receiver at a later point in time. The receive operation gets blocked if no data is available in the buffer.

Finally, the *amount of data* is defined as the number of bytes that must be transmitted from sender to receiver. The data can be sent at once or at various times during the communication interval.

2.3 Example

Figure 3 illustrates an application being composed of three tasks t_1, t_2, and t_3, each refined by components.

Task t_1 is composed of two components k_1 and k_2. The number of instructions for k_1 is $|k_1|$. During its execution, it communicates with component k_3 of task t_2. The starting time of the communication is defined by P_1 for k_1, so that k_1 potentially communicates between this time and the end of k_1. After having executed the instructions of k_1, the instructions of the second component k_2 will be executed (since $k_1 \rightarrow k_2$). No communication relation is defined for k_2. The components k_3, k_4, k_5 and k_6 as well as their relationships are defined in an analogous way for tasks t_2 and t_3.

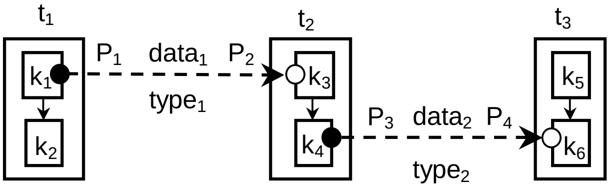

Figure 3: Example of an application

So far, we defined an application model that enables us to refine the tasks in the model using the concept of components and to define communication relations between components. In our example task t_2 communicates with task t_1 and afterwards with task t_3, so the structure and behavior of task t_2 is better understood using components than without refinement.

3. Reservation Model

The reservation model provides the basis for the calculation of the reservation time for the application. The enhanced application model implies a reservation model, where the refined structure of a task is used to estimate the reservation time more precisely.

3.1 Formal Definition

The reservation model is defined as an extended graph $G = (P, E, C, \gamma, \delta)$, where:

$P = \{p_1, p_2, \ldots, p_n\}$ is a set of vertices that represent component partitions of the tasks in the application model. Each leaf component k_i in the component hierarchy of a task is mapped onto exactly *one* partition p_i, iff k_i has *no* communication relations to other components. A leaf component k_i is mapped onto exactly *two* partitions p_{i1} and p_{i2} iff there exists *at least one* communication relation to another component.

E: PxP is the set of directed edges that represents the order in which component partitions are executed. We define $(p_i, p_j) \in E$ iff p_i represents component k_i and p_j represents component k_j and $k_i \rightarrow k_j$ or iff p_i and p_j represent a communicating component partitioned into p_i and p_j.

C: PxP is another set of directed edges that represent the communication relation between components. If a component k_i communicates with component k_j, k_i is mapped onto two partitions p_{i1} and p_{i2}, where $(p_{i1}, p_{i2}) \in E$, and component k_j is mapped onto p_{j1} and p_{j2}, where $(p_{j1}, p_{j2}) \in E$. Communication is represented by the edge $(p_{i2}, p_{j2}) \in C$, where p_{i2} is the sender and p_{j2} is the receiver.

γ: V → *instr.* is a marking of partitions that represents the number of instructions a partition p_i inherits from component k_i, which it represents. If partition p_i represents *one* component k_i, then p_i is marked by the number of instructions $|k_i|$. If *two* partitions p_{i1} and p_{i2}, where $(p_{i1}, p_{i2}) \in E$, represent a component k_i, p_{i1} is marked by the number of instructions where definitely *no* communication takes place. Partition p_{i2} is marked with the number of instructions, where k_i *potentially* communicates defined by the maximum of all functions P_j of all communication relations, in which k_i is involved. By marking p_{i2} as described, we make a maximum estimation that allows us to calculate the reservation time of the corresponding task.

δ: C → *data* is the weight of the edges in C. The weight represents the amount of data defined by the communication relation between two components k_i and k_j. The type of communication is not considered in this reservation model, so that we currently treat asynchronous communication like synchronous communication, which is a more restrictive approach and reduces performance gains if throughput is to be considered, as the only metric.

3.2 Example

The example of Figure 3 is transformed in the reservation graph shown in Figure 4.

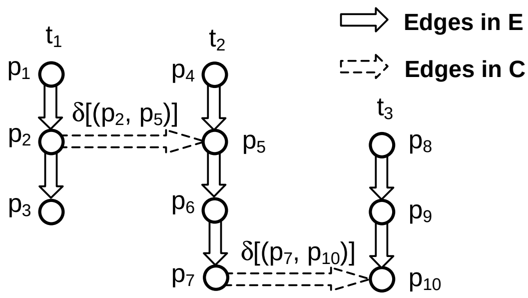

Figure 4: Example of a reservation graph

If we consider task t_1 in more detail, the first component k_1 of t_1 that communicates with the first component k_3 in t_2, is mapped onto two partitions p_1 and p_2, which are connected by an edge of E. Partition p_1 is marked with $\gamma(p_1)$, the number of instructions of k_1 that are not involved in communication with t_2. Partition p_2 is marked with $\gamma(p_2)$, the number of instructions where communication potentially takes place. The edge between p_2 and p_5 represents the communication between k_1 and k_3. The weight of the edge $\delta[(p_2, p_5)]$ is the amount of data to be transferred. The second subcomponent k_2 of t_1 is mapped onto partition p_3, which is connected to partition p_2 with an edge of E. Partition p_3 is marked with the whole number of instructions $|k_2|$ of k_2, because k_2 has no communication relation to other components.

We define the rest of the graph in a similar way.

4 Reservation time

Distributed resources in Grid computing are co-allocated [6] to be available for an application at the same time. We assume each task of an application to start at the same time using this co-allocation mechanism.

We consider the longest execution time of each task, which means that for each task the worst case is taken into account to estimate the longest execution time for the application. The execution time of a task is computed using the reservation graph and the execution times of component partitions in the graph.

The component partitions of a task in the reservation graph must be divided in two groups: partitions that have communication relations to other partitions, represented by edges in C, and partitions without such a communication relation.

The reservation time for a partition p_i without a communication relation, is simply computed by the marking $\gamma[p_i]$ of the partition divided by the MIPS of the resource the task t_j, containing partition p_i, is scheduled on. The reservation time of a partition without a communication relation is abbreviated as $\gamma_{tj}[p_i]$.

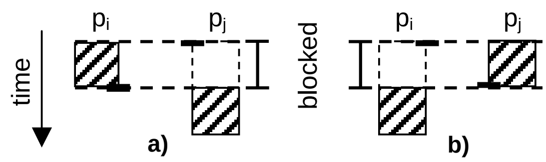

Figure 5: communicating tasks

To calculate the reservation time of a partition p_i that participates in a communication (an edge in C) with partition p_j, we must consider the markings $\gamma[p_i]$ and $\gamma[p_j]$. The marking declares the set of instructions, where one of these instructions is the first send operation (perhaps followed by further send operations) on the sender side, or the first receive operation on the receiver side. Two extreme scenarios are possible, as shown in Figure 5.

In Figure 5 a) the last instruction in p_i is a send operation, whereas the receive operation is the first operation in p_j. The receive operation will be blocked until the send operation is called, which means p_j has to wait until p_i reaches its end. In Figure 5 b) the send operation is called as the first instruction of p_i and the receive operation is called as the last instruction of p_j. The result will be a blocked sender p_i until p_j executes the receive instruction. As both scenarios show, we have to sum all execution times $\gamma_{tk}[p_k]$ of the partitions p_k that have a communication relation to other partitions, because the represented components can be blocked by the other components until they call their communication operations.

Further, we assume blocking communication primitives, so that p_i and p_j are blocked while data is transferred. Therefore, the time for data transfer is added to the execution time of the instructions. Communication time is calculated by dividing the marking $\delta[(p_i, p_j)]$ of the edge in C by the bit rate of the communication channel between the resources, onto which tasks t_i and t_j are mapped. The resulting communication time is shortened by $\delta_{ti,tj}[(p_i,p_j)]$. These communication times are added to the execution time of the instructions. The resulting reservation time is assigned to each partition.

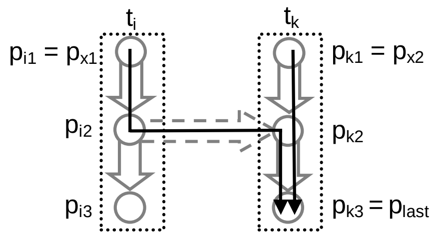

Figure 6: Paths through the graph

To compute the reservation of a task t_k we must consider all possible paths $w_s = (p_x, \dots, p_i, p_j, \dots, p_{last})$ from each first partition p_x of any task of the application to the last partition p_{last} of task t_k using edges in set E or C (Figure 6). Thus, the constraints

$\neg (\exists\, p_y \in P\ [\ (p_y, p_x) \in E])$,

$\neg (\exists\, p_y \in P\ [\ (p_{last}, p_y) \in E])$, and

$(p_i, p_j) \in E \lor (p_i, p_j) \in C$

for each pair p_i, p_j in path w_s must hold.

All paths from the first partition of any task in the application including t_k have to be considered, because the tasks are started at the same time using co-allocation, and, due to the communication relation, can influence the execution of task t_k.

The reservation time $R[w_i]$ of a path w_i is computed by estimating the reservation time for each partition as described above and summing up all these times. If partitions in the path are connected by edges in C, only the estimated time of one partition is taken into account, because that time already considers the other partition.

The reservation time or total completion time $T_{CT}[t_k]$ of a task t_k is computed by the maximum of all reservation times $R[w_i]$ of all paths w_i:

$T_{CT}[t_k] = \mathrm{Max}(\ R[w_i]\ )$ for all ($k \leq$ number of tasks), ($i \leq$ number of paths)

The total completion time T_{CT} of the application is computed as the maximum of all completion times $T_{CT}[t_k]$ of tasks t_k:

$T_{CT} = \mathrm{Max}(\ T_{CT}[t_k]\ )$ for all ($k \leq$ number of tasks)

4.1 Example

If we consider the reservation time of task t_3 in Figure 4, we must compute all paths from each first partition p_1, p_4, and p_8 of task t_1, t_2, and t_3 to p_{10}, the last partition of task t_3. The resulting three paths using edges in E and set C are:

$w_1 = (p_1, p_2, p_5, p_6, p_7, p_{10})$,
$w_2 = (p_4, p_5, p_6, p_7, p_{10})$, and
$w_3 = (p_8, p_9, p_{10})$,

In the computation for these paths, p_{10} and p_7, as well as p_5 and p_2 are connected with an edge in C. We must sum the estimated reservation time of these partitions only once, because the estimated reservation times are based on each other, as described above.

In the reservation time of w_2, we take $\gamma_A[p_2]$ and $\delta_{A,B}[(p_2,p_5)]$ into account even though p_2 is not explicitly in w_2, because partition p_5 has a communication relation with p_2 and therefore can be blocked during execution time of p_2. That applies to $\gamma_B[p_7]$ and $\delta_{B,C}[(p_7,p_{10})]$ in the reservation time of w_3 too.

Therefore the reservation time R of each path w_i is:

$$R[w_1] = \gamma_A[p_1] + \gamma_A[p_2] + \gamma_B[p_5] + \delta_{A,B}[(p_2,p_5)] + \gamma_B[p_6] + \gamma_B[p_7] + \gamma_C[p_{10}] + \delta_{B,C}[(p_7,p_{10})]$$

$$R[w_2] = \gamma_B[p_4] + \gamma_B[p_5] + \gamma_A[p_2] + \delta_{A,B}[(p_2,p_5)] + \gamma_B[p_6] + \gamma_B[p_7] + \gamma_C[p_{10}] + \delta_{B,C}[(p_7,p_{10})]$$

$$R[w_3] = \gamma_C[p_8] + \gamma_C[p_9] + \gamma_C[p_{10}] + \gamma_B[p_7] + \delta_{B,C}[(p_7,p_{10})]$$

The maximum of these times is the reservation time T_{CT} of task t_3:

$$T_{CT}[t_3] = \mathrm{Max}(R[w_1], R[w_2], R[w_3])$$

To compute the total completion time T_{CT} of the application we compute the maximum of the completion times of all tasks:

$$T_{CT} = \mathrm{Max}(T_{CT}[t_1], T_{CT}[t_2], T_{CT}[t_3])$$

We note that until the tasks are not mapped to resources, the reservation time cannot result in a specific value. It is not allowed to sum the instructions of different tasks, because tasks are scheduled on different resources and the instructions of tasks are executed on these resources at different speeds. Although the value is not defined, the calculation of the reservation time can be done once for an application, which means that it must not be done during scheduling.

5. Scheduling Heuristic

The scheduling problem that maps tasks to resources is known to be NP-complete. To derive an optimal schedule, all combinations of task-resource pairs for the whole application must be considered. Thus, heuristics are examined to find schedules that reach acceptable performance for a certain class of applications.

We propose a heuristic for the enhanced application model, which takes advantage of the refined task model and the known communication dependencies. The heuristic is embedded into the Greedy algorithm shown in Figure 7.

The tasks are ordered by decreasing number of instructions (line 1.), because the largest task should be mapped to a fast resource first, so that the impact of the instructions of that task on other tasks due to communication relations, is minimized.

1. order application tasks by decreasing number of instructions
2. initialize all $\gamma_{ti}[p_k] = 0.0$ in T_{CT}
3. initialize all $\delta_{ti,tj}$ $[(p_k,p_l)] = 0.0$ in T_{CT}
4. initialize PLACEMENT$[t_i] = -1$ for all tasks t_i
5. for each task t_i
   - 5.1. for each resource R_k
     - 5.1.1. PLACEMENT$[t_i] = R_k$
     - 5.1.2. compute $\gamma_{ti}[p_x]$ in T_{CT}
     - 5.1.3. compute $\delta_{tm,tn}[(p_k,p_l)]$ in T_{CT}, where t_n and t_m already mapped
     - 5.1.4. compute T_{CT}
     - 5.1.5. Store best T_{CT} and best associated resource R_{best}
   - 5.2. end for
   - 5.3. PLACEMENT$[t_i] = R_{best}$; whereby R_{best} is not further available for other tasks.
6. end for

Figure 7: Enhanced heuristic

The algorithm is based on the reservation model and the resulting total completion time T_{CT} of the application. The execution time $\gamma_{ti}[p_k]$ and communication time $\delta_{ti,tj}[(p_k,p_l)]$ involved in the computation for T_{CT} are not defined if the task or tasks are not mapped to a resource. They are initialized to the value 0.0, so that they have no effect in the computation of the paths (lines 2. and 3.) until the corresponding task or tasks are mapped to specific resources.

Finally, the algorithm iterates over all tasks and resources (line 5. and 5.1). In each step, the current task is mapped onto the corresponding current resource and the execution times $\gamma_{ti}[v_k]$ and the communication times $\delta_{ti,tj}[(v_k,v_l)]$ are recomputed for that configuration. As a result of the new calculated execution times and communication times, the total completion time T_{CT} of the application is computed and compared to the best previous mapping. If the new mapping is better, the resource is assumed to be a better configuration for that particular task.

The algorithm is based on the recomputation of the total completion time T_{CT} in each step (line 5.1.2). The elements of T_{CT} are not defined completely until the last task is mapped onto a resource. We only estimate T_{CT} in each step by assuming that the elements of T_{CT} that are not computable in this step, have no effect in the calculation.

6. Experimental Results

To evaluate the advantage of the component-based approach introduced in the application model and the proposed heuristic, we compare our implementation to another heuristic that does not use information about the internal structure of tasks. We decided to compare our heuristic with the algorithm proposed by Jon Weissman [17], which uses communicating tasks without further refinement. We use the category "*concurrent*" of communicating task, which means, that tasks are blocked while data is transferred between them. We do not compare to the categories "*parallel*" or "*pipeline*" of the model, since our reservation model, as yet, does not support asynchronous communication. We used our reservation model to compare the completion time T_{CT} for both algorithms with the optimum T_{CT} reachable by the brute-force exponential algorithm.

To compare the results of both heuristics, the values of our evaluation environment are chosen from the environment proposed in [17].

6.1 Weissman's Heuristic

Jon Weissman proposed a Greedy-algorithm, which is based on the calculation of T_{CT} (Figure 8). The algorithm iterates the tasks and resources and computes the total completion time T_{CT} in each step. The calculation of T_{CT} is based on the total number of instructions of tasks and the amount of data transferred between them.

1. order application tasks by decreasing comp_amt value
2. initialize PLACEMENT[C_i]=-1 for all tasks C_i
3. for each component C_i
   - 3.1. for each resource S_k
     - 3.1.1. compute T_{CT} with PLACEMENT[C_i] = S_k, given PLACEMENT[C_j], i < j, unchanged
     - 3.1.2. remember best T_{CT} and best associated site S_{best}
   - 3.2. end for
   - 3.3. PLACEMENT[C_i] = S_{best}
4. end for

Figure 8: Weissman's algorithm

To compare the algorithms we transform our application model into the model of Jon Weissman. Therefore, the number of instructions of a task is computed by summing all leaf components in the component hierarchy of a task. The data transferred between two tasks is computed by summing the amount of data of all communication relations between them.

6.2 Evaluation Environment

The evaluation environment for both algorithms is defined by the parameters used to generate the application model and the interconnected resources.

Description	Value range
Tasks	3 … 8
Components	2 … 5
Instructions	[10000, 100000]
Data (Bytes)	[50000, 150000]

Table 1 – Task settings

As shown in Table 1, the application model covers 3 to 8 tasks (dependent on the results to be evaluated) each containing 2 to 5 components assuming they are uniformly distributed. We do not refine a task into further subcomponents, but show that even the refinement in 2 to 5 components results in better schedules. The number of instructions of a tasks which is the sum of the number of instructions of the components, range between 10000 and 100000 instructions chosen randomly from the interval using a normal distribution. Communication is randomly generated between components that transfer data between 50000 and 150000 Bytes. The amount of data is also normally distributed. The parameter a_i of function P_i used in the communication relation is uniformly generated between 0 and 1.

Description	Value range
Resources	3 … 10
Speed (MIPS)	[1, 10000]
Connection (Kbps)	[100, 1000]

Table 2 – Resource settings

Table 2 shows the parameters of the resources generated. We evaluated the algorithms using 3 to 10 resources (dependent on the results to be evaluated). The speed of the resources range from 1 to 10000 MIPS, also generated from a normal distribution. The generated resources are interconnected by links with bit rates between 100 and 1000 Kbps normally distributed. The resources are assumed to be fully connected.

The values of the application model and the resources serve the basic comparison of the algorithms shown in Figures 7 and 8.

6.3 Results

For each measuring point of the algorithms in the following diagrams, we computed 10.000 different environments and compared their schedules.

Our first evaluation is performed with a fixed number of 10 resources and an increasing number of tasks ranging from 1 to 8. Figure 9 shows the results of the performed algorithms for these values.

The increasing number of tasks is shown on the x-axis, while the y-axis shows the percentage relative to the optimal schedule, assigned to be 100%.

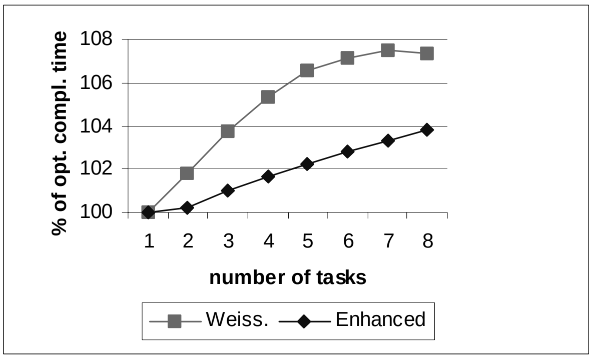

Figure 9: increasing number of tasks

If only one task is considered, both algorithms reach the best possible schedule. Since we consider only one application in the system, both algorithms will map the task of the application onto the best resource. The more tasks that are added to the application, the more advantage can be taken from the information about the refined tasks and the enhanced application model. This is why our algorithm performs better than that of Weissman. The refined structure provides information about the dependencies of one task to the others. If we consider the reservation model, late communicating tasks are dependent on earlier communicating tasks if a path in the reservation model exists from the involved partitions of the communication to the partitions involved earlier in communication, which can be proven by the paths through the graph. Our algorithm takes dependencies into account, but the algorithm of Weissman cannot, because it has no information about these dependencies. So our algorithm results in better schedules for these cases too.

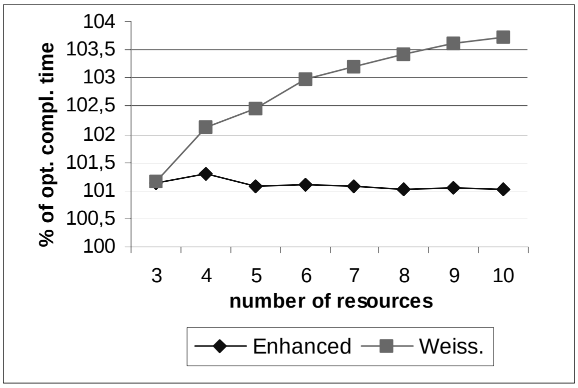

Figure 10: Increasing number of resources

Our second evaluation is performed with a fixed number of 3 tasks and an increasing number of 3 to 10 resources. Figure 10 shows the results of that scenario.

The increasing number of resources is shown on the x-axis, while the y-axis shows the percentage relative to the optimal schedule, assigned to 100%.

If we consider three resources, both algorithms show nearly the same performance. As the number of resources increases, the information about the refined tasks can be used by the proposed algorithm to compute better schedules for the application. The enhanced algorithm is more robust against increasing number of resources in the system, because the dependencies of the tasks are used to evaluate the resources in more detail even if new resources are available in the system.

The following related work delivers insight in currently used application models in Grid Computing. The enhanced application model is compared to these models in short.

7. Related Work

Application models in Grid computing can be classified as follows:

- Single Task Model
- Directed Acyclic Graph (DAG) Model
- Meta application Model

The single task model [11] defines an application to be a set of independent tasks that do not communicate with each other. A task can be executed without taking the other tasks into account, so it can be scheduled independently. Each task of the model defines the number of instructions to be executed, so that the scheduler can estimate the execution time of that task on a specified resource. A refined task structure is not given in the model and communication cannot be modeled. Research in the area of the single task model focused on the characteristics of scheduling algorithms for these models in Grid environments [8] [13]. A special group of single task models are parameter sweep applications [4] that define long running tasks, where each task defines a set of input files containing data to be processed. A single file might be input to more than one task. A refinement of tasks is not given in that model, either.

The DAG model [10] uses edges to define the execution order of tasks. Each task defines the number of instructions and communication is represented as the weight of the edges defining the data transferred asynchronously between tasks. The tasks of the model cannot be refined in structure and concurrent execution is not possible between communicating tasks. It is possible to model the DAG in our proposed application model using asynchronous communication, but the DAG model is not as expressive as ours, due to the missing synchronous communication concept. Research in the field of DAGs in Grid environments is also focused on scheduling algorithms [1] [5].

The meta-application model proposed in [17], defines a set of communicating tasks. Communication dependencies are divided into three categories: pipeline, concurrent, and parallel. The category "pipeline" is equivalent to the relations between tasks in the DAG model. It specifies the order in which tasks are executed. The category "concurrent" specifies that tasks synchronize on the data transfer, and therefore block during transmission. The category "parallel" specifies that communicating tasks are not blocked during data transfer, but tasks that communicate, must run in parallel. Tasks of the model are defined by the instructions to be executed, but a refinement of tasks is not possible and the point in time that defines when communication potentially occurs is not included in the model. Our model enhances the model by refining the structure of tasks and defining the communication relation between components.

8. Conclusion and Future Work

Developers of applications such as simulations can provide information about the internal structure and the behavior of the application. We propose an enhanced application model, where knowledge about the application can be mapped into components within tasks and communication primitives. We also propose a reservation model, including synchronous communication, and a heuristic thus using this additional information in the model. Experimental results compared to a heuristic based on an application model without that information showed that the new heuristic results in better schedules.

Further research will focus primarily on the reservation model, where fine-grained knowledge about communication can further improve the resulting schedules. We will introduce asynchronous communication in the model and propose heuristics that combine the knowledge about synchronous and asynchronous communication to enhance schedules. This also implies further research on the heuristic, which should adapt to the information contained in the application model.

References

[1] A. H. Alhusaini, V. K. Prasanna, C.S. Raghavendra, "A Unified Resource Scheduling Framework for Heterogeneous Computing Environments", *Proceedings of the 8th IEEE Heterogeneous Computing Workshop*, Puerto Rico, 1999, pp. 156 - 166.

[2] C. Baillie, J. Michalakes, R. Skalin, "Regional Weather Modeling on Parallel Computers", *Special Issue of Parallel Computing*, Vol. 23, No. 14, 1997, pp. 2135 – 2142.

[3] R. Buyya, S. Chapin, D. DiNucci, "Architectural Models for Resource Management in the Grid", *Proceedings of 1st IEEE/ACM International Workshop on Grid Computing*, 2000, pp. 18 – 35.

[4] H. Casanova, A. Legrand, D. Zagotodnov, F. Barman, "Heuristic for Scheduling Parameter Sweep Applications in Grid Environments", *Proceedings of the 9th IEEE Heterogeneous Computing Workshop*, Cancun 2000, pp. 349 – 363.

[5] H. Chen, M. Maheswaran, "Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems", *Proceedings of the 11th IEEE Heterogeneous Computing Workshop*, Fort Lauderdale, 2002, p. 88b.

[6] K. Czajkowski, I. Foster, K. Kesselman, "Resource Co-Allocation in the Grid", *8th IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, 1999, p. 37.

[7] J. Dongarra, G. Geist, R. Mancheck, V. Sunderam, "Integrated PVM Framework Supports Heterogeneous Network Computing", *Computers in Physics*, Vol 7, No. 2, 1993, pp. 166 – 174.

[8] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, A. Streit, "On Advantages of Grid Computing for Parallel Job Scheduling", *Proceedings of 2nd ISSS International Symposium on Cluster Computing and the Grid*, 2002, pp. 39 – 47.

[9] I. Foster, C. Kesselman, *The GRID: Blueprint for a new Computing Infrastructure*, Morgan Kaufmann, San Francisco, 1999.

[10] M. Iverson, F. Özgüner, "Dynamic, Competitive Scheduling of Multible DAGs in a Distributes Heterogeneous Environment", *Proceedings of the 7th IEEE Heterogeneous Computing Workshop*, Orlando, 1998, pp. 70 – 78.

[11] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, "Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems", *Proceedings of the 8th IEEE Heterogeneous Computing Workshop*, Puerto Rico, 1999, pp. 30 - 45.

[12] Message Passing Interface Forum. "MPI: A Message-Passing Interface standard." *International Journal of Supercomputer Applications*, 8 (3/4), 1994, pp. 165 – 414.

[13] R. Min, M. Maheswaran, "Scheduling Advance Reservation with Priorities in Grid Computing", *13th International Conference on Parallel and Distributed Systems*, Anaheim, 2001.

[14] K. Rangamnathan, I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications", *11th International Symposium on High Performance Distributed Computing*, Edinburgh, 2002, pp.352 – 359.

[15] J. Weidendorfer, P. Luksch, "A Framework for Transparent Load Balancing in Parallel Numerical Simulation", *34th Annual Simulation Symposium*, Seattle, 2001, pp. 125 – 132.

[16] J. Weissman, X. Zhao, "Scheduling Parallel Applications in Distributed Networks", *Journal of Cluster Computing*, Vol. 1, No.1, 1998, pp. 109 – 118.

[17] J. Weissman, "Scheduling Multi-Component Applications in Heterogeneous Wide-area Networks", *Proceedings of 9th IEEE Heterogeneous Computing Workshop*, Cancun, 2000, pp. 209 – 215.