Guiding Performance Tuning for Grid Schedules

Jörg Keller Wolfram Schiffmann
FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Postfach 940, 58084 Hagen, Germany
{joerg.keller,wolfram.schiffmann}@fernuni-hagen.de

Abstract

Grid jobs often consist of a large number of tasks. If the performance of a statically scheduled grid job is unsatisfactory, one must decide which code of which task should be improved. We propose a novel method to guide grid users as to which tasks of their grid job they should accelerate in order to reduce the makespan of the complete job. The input we need is the task schedule of the grid job, which can be derived from traces of a previous run of the job. We provide several algorithms depending on whether only one or several tasks can be improved, or whether task improvement is achieved by improvement of one processor.

Keywords*: Task scheduling, Grid computing, Performance Tuning*

1. Introduction

In the ongoing run between resources available to solve large computational problems and computational requirements demanded from applications that want to solve larger instances of such problems, grid computing has opened a new stage beside the topmost supercomputers from the TOP500 list. While the latter can only be afforded by few computing centres in the world, computational grids offer the chance to fulfil the demands above by collaborative efforts of several not-so-wealthy sites.

Applications running on such grids (so-called *grid jobs*) have to exploit the parallelism in the grid resources, and together with the complexity of the problems to be attacked often lead to grid jobs consisting of hundreds or thousands of tasks. While a grid job may be carefully planned, implemented, and statically scheduled onto a grid that it can use exclusively for execution, the performance may still be unsatisfactory, without any obvious bottleneck. Traces from such runs can be used to update the task graph parameters used for the scheduling, but they may also be used to identify bottlenecks, i.e. reasons why the performance is worse than planned. Because of the size of these traces, this cannot be done by hand, and hence tools are necessary to support the grid users in this task.

In the present work, we treat the problem above algorithmically. Given the task graph of a grid job reflecting its mapping, which can be derived from the trace of a previous run, we analyze this graph and compute one or several tasks where a runtime improvement of a certain percentage would lead to the maximum runtime improvement for the complete grid job. Thus, programmers get recommendations which code they should look at first. We discuss several variants that differ in the number of tasks to be changed, i.e. adapt to available programmer resources, in the amount of computation necessary, and in the amount of improvement that the promise to offer. We illustrate our algorithms with an example.

The remainder of this article is organized as follows. Section 2 summarizes related work. Section 3 formalizes the problem to be solved. Sections 4 and 5 presents algorithms to recommend one or several tasks for improvement, respectively. In Sect. 6, we discuss several variations of the problem and possible algorithmic solutions. Section 7 provides a conclusion and an outlook on further work.

2. Related Work

Scheduling of the tasks on the Grid may require some criteria to be considered. Besides the execution time, or makespan, which is used most often also efficiency, economic costs, reliability and quality of service can be of importance. There is an huge number of related work in the area of grid scheduling. It can be categorized with respect of the number of simultaneously scheduled grid jobs, the scheduling process, the task model, resource model and the critera used [5]. The scheduling process can manage single or multiple grid jobs simultaneously, it can be static or dynamic, use single or multiple criteria. Also there could be schedules with and without advance reservation. Task must

be mapped on resources. This mapping can be variable or fixed. It is also possible that the tasks migrate from one resource to another. The resources (processing elements PE) in a grid are usually heterogeneous but they can also be partly homogeneous, e.g. a compute cluster.

Analysis of event traces in grid environments shows that it is nearly impossible to explain the occurrence of wait states during the execution of grid jobs. Usually, the symptoms and its cause are separated by great temporal or spatial distances. Hermanns et al. [3] proposed a method for verifying hypotheses on distant performance phenomena, which is based on MPI event traces. By simulating the message-passing behavior recorded in the event traces one can verify hypotheses without altering the application. Unfortunately, one has to guess about possible causes because the approach isn't able to identify them automatically. Often, the main objective of trace-based approaches isn't the identification of bottlenecks or improvements of a specific grid job. Instead, one tries to predict the scalability of an application on bigger target computing systems [1]. Some well known scalable performance-diagnosis tools are KOJAK and SCALASCA [3, 6].

3. Analyzing Grid Job Traces

A grid job consists of a set V of n tasks, represented as a connected, directed acyclic task graph $G = (V, E)$ where each node represents a task and each edge (u, v) represents a communication upon completion of task u, necessary to start task v. Each task is associated a runtime, i.e. there is a mapping

$$\rho : V \to \mathbf{R}^+ ,$$

and each edge is associated a communication time, i.e.

$$\varepsilon : E \to \mathbf{R}^+ .$$

We introduce two additional nodes with runtime 0 in V, a start node v_s and an end node v_e. There are edges with communication time 0 from v_s to every task with indegree 0, and to v_e from every task with outdegree 0. This serves to have unique source and sink nodes.

We assume that beyond the task graph G, there exists a static schedule onto an underlying grid hardware which is exclusively used to execute the grid job under consideration. If we do not possess the task graph itself, we can derive it from traces generated during a run of the grid job [3].

Our assumptions are rather exact but we think that they are not as unrealistic as they seem. If one would work in a completely dynamic grid infrastructure, then no static scheduling would be of any worth. Yet, many grids allow reservation in advance, so that the infrastructure might not be exclusively used by the grid job under consideration, but as there performance guarantees from the reservation it is known which compute resources are available. Also, while the exact runtime of any task may depend on the particular input data, the runtime is typically bounded by the algorithm implemented in that task. If there is a strong dependence of the task's runtime on the input data, then such a grid job might not be suited to static scheduling or advance reservation.

We did not mention heterogeneity of resources in particular but assume that it is handled by the static scheduler of the grid job, and is implicitly expressed in the task runtimes. When we reduce the runtime of the grid job, we still want to maintain the mapping of tasks onto processors to be able to restrict our attention to task graphs. Therefore we represent the mapping indirectly in our task graph by inserting edges (u, v) with communication time 0 if tasks u and v are mapped to the same processor and scheduled successively. While there may be some cases where an improved schedule may lead to further improvements by re-mapping tasks onto heterogeneous resources, we feel that this is not very frequent and thus should be treated separately instead of complicating our current proposal.

In our scheduling model, a task can only be started if all tasks that communicate to it are completed, and the communication done. The job is completed if all tasks are completed. Thus, we can compute for each task $v \in V$ the earliest time $b(v)$ when the task can be completed. This value is called the *static b-level*. Obviously $b(v_s) = 0$ and for $v \neq v_s$:

$$b(v) = \rho(v) + \max\{b(u) + \varepsilon(u, v) \mid (u, v) \in E\} . \quad (1)$$

Then, the earliest time when the job can be completed is $b(v_e)$, called the *makespan*. As we have represented the mapping to processors via edges, the makespan computed for this task graph is identical to the makespan of the underlying schedule.

The b-levels can be computed as follows. Initially, we set $b(v) := 0$ for all nodes in V, and sort the graph G topologically, i.e. we arrange it in layers such that v_s is the top most layer, v_e is the bottom most layer, and all edges move from an upper layer to a lower layer. Figure 1 gives an example.

Then, in this order, i.e. layer by layer, and node by node in the layer, for every successor v of node u in the layer, we compute

$$b(v) := \max\{b(v), b(u) + \varepsilon(u, v) + \rho(v)\} .$$

After processing the graph, Eq. (1) holds.

Obviously, by re-defining the edge weight as

$$\varepsilon' : E \to \mathbf{R}^+, \quad \varepsilon'(u, v) = \frac{1}{1 + \varepsilon(u, v) + \rho(v)}$$

one could get rid of the node weights ρ, and computing $b(v_e)$ became equivalent to solving a single-pair (or equivalently a single-source) shortest-path problem in a directed

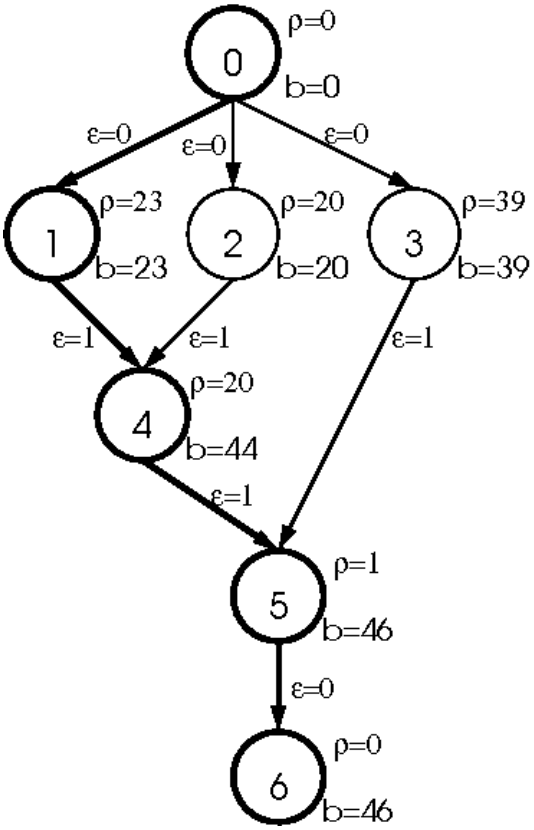

Figure 1. Example task graph, sorted topologically.

acyclic graph, see e.g. [2], which can be done in time $O(|V| + |E|)$.

Typical solutions for this problem not only compute the b-level for each node, but also the so-called *critical path* of nodes $v_s = v_0, \ldots, v_k = v_e$ that determines the b-level of the sink node v_e, and thus the execution time of the job. Note that the critical path need not be unique, but that we will assume a unique critical path in the sequel for simplicity. In our example of Fig. 1, the critical path is marked in bold, comprising nodes 0, 1, 4, 5, and 6.

As we want to guide the improvement of the makespan by acceleration of task code, we assume that communication times are fixed. Furthermore, we make the assumption that the runtime of any task v cannot be improved more than a given fraction $t_{max} \in [0 : 1]$. Typically t_{max} will be on the order of less than 10%.

4. Finding the best task for improvement

As task graphs can be quite large, especially those generated from traces, and as performance tuning is time-consuming, the programmer needs a hint which tasks he should look into. More exactly, our first problem to solve is to select the task $v_{max} \in V$ where a runtime improvement by a fraction $t \leq t_{max}$ generates the maximum runtime improvement for the whole grid job, i.e. the maximum reduction of the makespan. The following Lemma 1 gives a hint as to which tasks to consider for improvement.

Lemma 1 *The task v_{max} is a task v_i on the critical path from v_s to v_e, before and after the runtime reduction.*

Proof: For any task v not on the critical path, a reduction of $\rho(v)$ does not influence the b-level of v_e, by the definition of the critical path. Thus, v_{max} must be on the current critical path. If v_{max} would not be on the critical path anymore after the runtime reduction, then we could have improved it by a factor $t' < t$, such that it would have remained on the critical path. After v_{max} leaves the critical path, further runtime savings do not improve the makespan. ∎

Lemma 1 indicates that the makespan is improved by improving the runtime of any task v_i on the critical path, given that this task still remains on the critical path after its improvement. We therefore restrict to tasks v_i with $1 \leq i \leq k-1$ because v_0 and v_k were added later on with runtime 0.

For each of these tasks v_i, we seek the maximum runtime improvement such that v_i does not leave the critical path and that the factor of improvement is not more than t_{max}. Then we recommend the task v_i promising the largest runtime improvement as candidate v_{max} for tuning the grid job.

Obviously, one may choose t_{max} individually for each task v_i, thus being able e.g. to mark or exclude a task v_i that

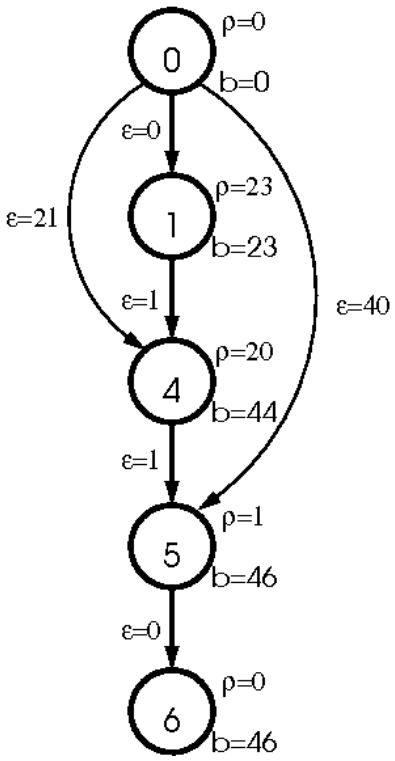

Figure 2. Line graph of example task graph.

has already been optimized by setting its t^i_{max} to a smaller value than others or even to 0%.

To compute when a task v_i will leave the critical path, we transform the task graph into a line graph $G' = (V', E')$. This graph only consists of the nodes v_i of the critical path ($i = 0, \ldots, k$) with runtimes $\rho_i = \rho(v_i)$. The nodes are linked by the edges (v_i, v_{i+1}), where $0 \leq i \leq k-1$, with weights $\varepsilon_{i,i+1} = \varepsilon(v_i, v_{i+1})$. The remainder of the original task graph that remains unchanged, is represented by edges (v_i, v_j), where $0 \leq i \leq k-2$ and $i+2 \leq j \leq k$, with edge weight $\varepsilon_{i,j}$ being the weight of the heaviest path from v_i to v_j that does not meet the critical path except in v_i and v_j, if such a path exists.

Figure 2 depicts this transformation for the task graph of Fig. 1. Only the nodes 0, 1, 4, 5, 6 from the critical path remain. There are only two paths outside the critical path: one from node 0 via node 2 to node 4 with weight $21 = 0 + 20 + 1$, and one from node 0 via node 3 to node 5 with weight $40 = 0 + 39 + 1$.

One can show that both critical paths mentioned in Lemma 1 are identical. As a consequence, v_i would leave the critical path as soon as v_i does not constitute anymore the maximum when computing $b(v_{i+1})$ in Eq. (1). Furthermore, as reducing the runtime of task v_i reduces the b-level of v_{i+1}, this also reduces the computation of the b-levels of v_{i+2} to v_k. Therefore, the same check has to be done for them as well.

Algorithmically, we proceed as follows. For each node v_j, where $i+1 \leq j \leq k$, we compute

$$\begin{aligned} d^i_j &= b(v_{j-1}) + \varepsilon(v_{j-1}, v_j) \\ &\quad - \max\{b(v_h) + \varepsilon(v_h, v_j) \mid h < i \text{ and } (v_h, v_j) \in E'\} \end{aligned} \tag{2}$$

Thus, d^i_j is the amount that the runtime of v_{j-1} can be reduced before the critical path condition is violated in v_j.

Note that it is not sufficient to subtract

$$\max\{b(u) + \varepsilon(u, v_j) \mid (u, v_j) \in E, u \neq v_{j-1}\} \tag{3}$$

which one would assume to be suggested by Eq. (1). If all paths from the source v_0 to u pass the critical path in one of the nodes $v_i, \ldots, v_{j-2}$, then a runtime reduction in v_i would also necessarily reduce the b-level of u accordingly, so that it cannot count as a constraint.

Obviously, the set in Eq. (2) over which the maximum is formed may be empty. In this case, we set $d_j = \infty$, because no runtime reduction of v_{j-1} can violate the critical path condition in v_j.

Then we compute, for $i = k-1, k-2, \ldots, 1$,

$$D_i = \min\{d^i_j \mid j > i\} \; .$$

Thus D_i is the amount that the runtime of task v_i can be reduced without violating the critical path condition in any of $v_{i+1}, \ldots, v_k$.

Finally, for each v_i, where $1 \leq i \leq k-1$, we compute the possible runtime reduction as

$$r_i = \min\{D_i, t^i_{max} \cdot \rho(v_i)\}$$

in order to avoid cases where e.g. a node v_i is on any critical path no matter how much its runtime would be reduced and hence would be proposed as a candidate v_{max} although large runtime improvements are not realistic.

Our recommendation is to consider task v_i with maximum r_i as candidate v_{max} for inspection of the code with respect to performance improvement. Of course, we could also give a recommendation list with all tasks of the critical path sorted with descending r_i values.

The computational complexity consists of computing the b-levels and the critical path in time $O(|V|+|E|)$, the transformation into the line graph which constitutes a kind of all-pair shortest-path problem (restricted to all pairs of critical path nodes), and the computation of the D_i which takes time $O(k^3)$.

In our example of Fig. 1, we compute $d^5_6 = d^4_6 = d^1_6 = \infty$ as the indegree of task 6 is only 1. Furthermore we have $d^4_5 = d^1_5 = 45 - 40 = 5$ and $d^1_4 = 24 - 21 = 3$. From this we receive $D_5 = \infty$, $D_4 = \min\{\infty, 5\} = 5$ and $D_1 = \min\{\infty, 5, 3\} = 3$. Thus, if all $t^i_{max} = 15\%$, we obtain

$r_5 = \min\{\infty, 0.15 \cdot 1\} = 0.15$, $r_4 = \min\{5, 0.15 \cdot 20\} = 3$, and $r_1 = \min\{3, 0.15 \cdot 23\} = 3$. We see that in this example both tasks 1 and 4 can be chosen as candidates, because both runtimes can be improved by 3. Yet the reasons are different: the runtime of task 1 can be improved only by 3 because of critical path constraints, and to achieve this would only need an improvement by $3/23 \approx 13\%$, while the runtime of task 4 can be improved only by 3 because of the threshold of 15%, which is already quite large. Thus, one would probably recommend task 1 for inspection first.

Note that an additional edge $(1, 6)$ in the original task graph with weight 22 would have rendered $d_6^5 = d_6^4 = 1$ while $d_6^1 = \infty$ still, leading to $D_5 = 1$ and $D_4 = \min\{1, 5\} = 1$, while $D_1 = 3$ remains unchanged. If one would have used Eq. (3) instead, then d_6^1 would have become $1 = 46 + 0 - (23 + 22)$ and thus $D_1 = 1$ would have been the consequence, although $D_1 = 3$ is possible.

5. Improving Several Tasks

As an extension of this scheme, one might also consider to improve *several* tasks. We constrain ourselves to the case that the critical path is not changed by the improvement. In the case where only one task was improved, this was guaranteed. Here it is a restriction as we will see later in our example.

The condition that the critical path is not changed despite of runtime improvements r_i in nodes v_i holds as long as for each pair of critical path nodes v_i and v_j that are not adjacent (i.e. $j \geq i + 2$), no path between them in the remainder of the original task graph has more weight than the critical path between them:

$$\begin{aligned} \varepsilon_{i,j} \quad \leq \quad & \varepsilon_{i,i+1} + \rho_{i+1} - r_{i+1} \\ & + \varepsilon_{i+1,i+2} + \cdots + \rho_{j-1} - r_{j-1} + \varepsilon_{j-1,j} \end{aligned} \tag{4}$$

We achieve our solution by solving a linear optimization problem. We seek to maximize the sum R of the runtime improvements r_i, i.e.

$$R = \sum_{i=1}^{k-1} r_i$$

under the constraints that for $i = 1, \ldots, k-1$ we have $0 \leq r_i/\rho_i \leq t_{max}^i$, and for all $1 \leq i, j \leq k-1$ with $i + 2 \leq j$, Eq. (4) holds. Note that the r_i can be considered to be reals so that the solution of the problem is simpler as in the integer case.

We may want to prefer solutions that improve as few tasks as possible. To do this, we have to introduce binary indicator variables x_i for improved tasks with the constraint $x_i \leq r_i$ if we assume r_i to be integral. Thus, in an optimal solution we have $x_i = 0$ for tasks that are not improved. We extend R to

$$\tilde{R} = \sum_{i=1}^{k-1} r_i + \alpha \cdot \sum_{i=1}^{k-1} (1 - x_i) \tag{5}$$

so that each unchanged task increases the solution by α. The choice of α determines the importance of this second optimization goal.

The complexity of the linear optimization problem is $O(k)$ variables and up to $O(k^2)$ constraints, where $k \leq |V|$. Yet, in practice many of the $\varepsilon_{i,j}$ will be zero and thus those constraints can be removed. Also, the length of the critical path k will often be much shorter than the size of the graph. For example, if $k \leq \sqrt{|V|}$ then we have $O(\sqrt{|V|})$ variables and $O(|V|)$ constraints.

In our example, we would have to maximize the sum of the variables r_1, r_4, r_5 with the following constraints:

$$\begin{aligned} 0 \quad &\leq \quad \frac{1}{23} \cdot r_1 \leq 0.15 \\ 0 \quad &\leq \quad \frac{1}{20} \cdot r_4 \leq 0.15 \\ 0 \quad &\leq \quad r_5 \leq 0.15 \\ 21 \quad &\leq \quad 24 - r_1 \\ 40 \quad &\leq \quad 45 - r_1 - r_4 \end{aligned}$$

We see that the third constraint is independent of the others, and thus in an optimal solution $r_5 = 0.15$. Also, the fourth constraint is tighter than the first, giving us $0 \leq r_1 \leq 3$. Together with the second constraint $0 \leq r_4 \leq 3$ the fifth constraint $r_1 + r_4 \leq 5$ determines all optimal solutions to be of the sort $2 \leq r_1, r_4 \leq 3$ with $r_4 = 5 - r_1$. Thus, the maximum reduction of the critical path is $r_1 + r_4 + r_5 = 5 + 0.15 = 5.15$. The simplicity of this example comes from the fact that the critical path is short and that all edges outside the critical path have their origin in the artificial source (task 0).

If we optimize our example for function $\tilde{R}$ of Eq. (5), we see that already for $\alpha > 0.15$, a solution with $r_5 = 0$ will give a better target value, whereas $\alpha > 3$ will lead to either r_1 or r_4 being set to 0.

Note that if we allow the critical path to change, we can derive a larger reduction of the makespan with improving 3 tasks. We may improve task 1 with $r_1 = 3$, so that the paths 0–1–4 and 0–2–4 both have weight 21, excluding the runtime of task 4. Then we may improve task 4 with $r_4 = 3$. Now the path 0–1–4–5 (excluding the runtime of task 5) has weight 39, while the path 0–3–5 (excluding the runtime of task 5) has weight 40, and so is the new critical path. If we now reduce the runtime of task 3 by $r_3 = 1$, both paths have weight 39, and the improvement on the makespan is 6, while the maximum improvement without changing the critical path was 5.15. Thus, the variants of the next section

that do not enforce this restriction may lead to better results, yet perhaps at the price of a larger computational complexity or the lack of guarantee of a good solution. Therefore, the method of this section should always be applied as a reference first.

6 Variants

In the previous sections we proposed algorithms for identification of single or multiple tasks that mainly influence the performance of a grid job. In order to reduce the makespan exactly those tasks should be improved. Besides the proposed algorithms a great variety of other optimization algorithms could also be applied to achieve this objective. If just a performance function is available we could use heuristic search methods, e.g. meta-heuristic like Simulated Annealing, Evolutionary or Genetic Algorithms or even Ant Colony Optimization. Unfortunately, those algorithms perform an undirected search. Thus, it would be more advisable to use gradient search methods. To get the gradients we add some noise to the runtime $\rho(v_i)$ of task i which yields $\rho'(v_i)$. Then, we re-compute the static b-levels and get an approximation of the gradient as follows:

$$grad_i = \frac{b'(v_e) - b(v_e)}{\rho'(v_i) - \rho(v_i)}$$

Beginning with the initial state given by the event trace-based task graph we choose a step width η and perform a gradient descent to the next local minimum of the makespan.

Another variant could be to minimize the makespan by improving the performance of the resources instead of the tasks. While we transformed the event trace to a task graph the information about mapping of tasks to resources was ignored. Nevertheless, it is available and we could use it to determine which PE is most critical to the performance. Because many tasks are mapped to one resource, an improvement of one PE affects all those tasks. In principle, the above sketched gradient descent method can be transferred to solve this problem. Instead of approximating the gradients in terms of the runtime we have to compute an approximation of the gradient with respect to the performance of the resources.

A similar situation arises if the task graph contains copies of one task, e.g. the same code is executed in many nodes of the graph. In this case, the improvement of that code would affect many tasks at once. Nevertheless, note that the corresponding gradient components will be different and thus this case would be equivalent to the one described at the beginning of this section.

7. Conclusions

We have presented a novel method to guide the improvement of grid jobs by recommending, given the trace of a previous run, promising candidate tasks where an acceleration by a certain factor would maximize the gain on the makespan. We have detailed algorithms for finding a single candidate task and for finding a list of candidate tasks, thus adapting to the resources available for improvement on the side of the developer. So far, our algorithms concentrate on the critical path. We have indicated how our approach can be generalized by using meta-heuristics on the complete task graph to possibly find larger improvements, or to tackle related problems such as the question whether replacement of a single processor would result in a certain improvement. Our future work will concentrate on applying an implementation to both traces from real grid jobs [3] and to a benchmark suite of task graphs and schedules [4].

Acknowledgements

We would like to thank René Drießel for proposing the line graph representation, and Christoph Kessler for introducing us to the connection between scheduling problems and linear optimization.

References

[1] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. N. Wylie, and B. Mohr. Automatic trace-based performance analysis of metacomputing applications. In *Proc. International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[2] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] M.-A. Hermanns, M. Geimer, F. Wolf, and B. J. N. Wylie. Verifying causal connections between distant performance phenomena in large-scale message-passing applications. Technical Report FZJ-JSC-IB-2008-05, FZ Jülich, 2008.

[4] U. Hönig and W. Schiffmann. Fast optimal task graph scheduling by means of an optimized parallel A*-algorithm. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 842–848. CSREA Press, 2004.

[5] M. Wieczorek, A. Hoheisel, and R. Prodan. Taxonomy of the multi-criteria grid workflow scheduling problem. In *Proc. CoreGrid Workshop*, 2007.

[6] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient pattern search in large traces through successive refinement. In *Proc. Euro-Par*, pages 47–54, 2004.