Evaluation and Refinement of a Tuning Tool for Grid Applications

Jörg Keller, FernUniversität in Hagen, Dept. of Mathematics and Computer Science, 58084 Hagen, Germany
Ngoc Khanh Nguyen, FernUniversität in Hagen, Dept. of Mathematics and Computer Science, 58084 Hagen, Germany
Wolfram Schiffmann, FernUniversität in Hagen, Dept. of Mathematics and Computer Science, 58084 Hagen, Germany

Abstract

Tuning the performance of grid applications is cumbersome because it is very difficult to decide which code to look at. In our previous work we proposed a tool to recommend tasks in a task graph that look most promising for improvement. In the present work we first show how to extract task graphs from real grid applications with the help of SCALASCA, then we evaluate the tool with the OptSched benchmark suite of synthetic schedules. Finally, we present some extensions to the tool to adapt it to some typical situations occuring in practice.

1. Introduction

Grid applications often consist of very large code bases comprising hundreds of functions assembled from different sources. Hence, when performance of such a system is unsatisfactory, the question is what to do, i.e. which parts of the code to look at for improvement. For the case of data-parallel computing, data distribution can be changed to improve performance by reducing waiting times. A tool to advise users in this direction is developed [4] as part of the SCALASCA toolset [3]. However, for more unstructured types of large scale computations such as the Tachyon parallel raytracer [15] from the SPEC MPI2007 suite an acceleration may only be possible by improvement of part of the code.

In [8] we presented a concept for a tool to analyze grid applications with the help of task graphs. The tool selects one or several tasks from the critical path of a task graph such that the critical path remains unchanged, that no task needs to be improved by more than a certain percentage, and that the reduction in critical path length, i.e. makespan or runtime, is maximum given these constraints. In the present work, our contributions are as follows. We extend this concept in several ways. We show how it can be applied to traces of MPI applications, thus expanding its use beyond the range of grid workflows where task runtimes are known in advance. As an issue of immense practical importance, we incorporate the possibility of task types, i.e. the presence of several tasks based on the same code, so that an optimization will affect all of them. Also, we show how to treat the case of a non-unique critical path in the task graph. This case frequently appears as a result of a first optimization, so incorporation of this case allows to apply the tool iteratively. Finally, we evaluate our tool with a benchmark suite of synthetic task graphs and analyze the relationship of their characteristics and the findings of the tool.

In Section 2 we present how a grid application can be traced such that a task graph results that can be used with our tool, which thus becomes applicable to real world grid applications. Consequently, in Section 3 we can evaluate the tool with the OptSched benchmark suite of synthetic schedules [5, 6]. We present preliminary findings from this evaluation. In Section 4 we present two extensions of our tool: dealing with multiple instances of one type of tasks, where improvement of the code for this type reduces the runtime for all instances, and handling of a non-unique critical path by iterative application of our tool. In Section 5 we summarize our results and present an outlook to future work.

2. Task Graph Analysis

2.1. Original Guidance Tool

Our tool from [8] works in the following way. We assume that an application is given by a task graph. The task graph is a directed acyclic graph where each node represents a task, i.e. a piece of computation, that is assigned a runtime as node weight. Each task receives input only at its start, and provides output to other tasks only at its completion. The edges represent this information flow and are attributed with the time to communicate information from task to task as edge weight. If the tasks are both mapped onto the same processor, then we assume a zero communication time. We assume that the task graph has unique source and sink nodes. If it does not, we provide artificial source and/or sink nodes with zero node and edge weights.

Starting with a task graph already assigned to processing elements, we compute the critical path, i.e. the longest path from source to sink, which defines the makespan, i.e. the completion time executing the task graph. If we try to reduce the runtime by improvement of the code, obviously the tasks from the critical path must be considered, because other tasks do not directly influence the makespan. Also, we need improve a task on the critical path only so much that it remains on the critical path, a further improvement will not influence the makespan anymore. Finally, we assume that each task can only be improved by at most some

percentage t_m, because arbitrary improvements are unrealistic and because otherwise tasks that never leave the critical path (such as the source) could be considered as optimizable to a zero runtime. The value of t_m can either be set globally for all tasks, or individually for each task, e.g. 0% for highly optimized library routines, 10% for code from a master thesis, and 30% for untested code written in the night before.

Out of the tasks on the critical path, we select those that under the constraints above promise maximum makespan reduction, and recommend them for code review and improvement. The selection is done as follows. We transform the task graph into a line graph only consisting of the critical path. For each pair u, v of nodes on the critical path, we search for the longest path from u to v that does not touch the critical path, and represent this path by a so-called "outside" edge with the weight of the path. This is possible because we only change weights on the critical path. Now we generate a linear optimization problem, with the target runtimes of the critical path nodes as variables, the length of the critical path as objective function that is to be minimized, and the following constraints. Each target runtime must be between the original runtime and a fraction of $1 - t_m$ of the original runtime. Also, for each outside edge, the length of the critical path piece that it spans (i.e. the sum of the target runtimes of nodes on that critical path piece, plus the edge weights between) may never be smaller than the weight of the edge. In order to account for effort, we may provide a malus for each additional task that must be improved. So we get a balance between further improvement from recommending an additional task and further effort by having to inspect and improve the corresponding code.

2.2. Task Graph Extraction from Grid Applications

Task graph extraction from applications written in MPI has already been considered by Schulz [12]. Each piece of computation on a processor between two communication operations is a task. Each pair of a send and a receive operation leads to an edge between the tasks that the send terminates and the receives initiates, respectively. When runtimes of tasks are considered, waiting time in receives obviously is not modelled. A collective communication may be modelled with the help of sends and receives [1].

What must be considered additionally in our scenario are dependencies originating from the mapping of tasks onto processors, in particular the execution order of the tasks mapped on one processor. We represent these orders in the task graph by connecting the tasks mapped on one processor by a chain of edges with zero weight.

The SCALASCA toolset [3] allows to instrument an MPI application such that each processor writes a trace that logs function entry, function exit, send and receive operations. Thus, from these traces, with the method of Schulz augmented by the considerations outlined above, we are able to generate a task graph from the application such that the mapping is represented and such that we can associate each task with a function, i.e. with a piece of source code that the function executes. For functions with multiple communication operations, we can still associate with the i-th task since the function entry the source code before the i-th communication operation, assuming that this position can be found by static analysis in the source code.

2.3. Grid Workflows

In the previous subsection we have seen how an MPI application can be tuned by analyzing its task graph, and how this task graph can be extracted from a trace of the MPI application. MPI applications mostly run on cluster computers in order to provide fast communication of intermediate results on a fine grain level, although there are several projects like GridMPI [9] and MPICH-G2 [7] that provide grid-enabled implementations of MPI, and there are MPI applications running on grids [11]. Yet, grid environments are not well-suited for very fine grained parallelism. Instead, they are often used for high throughput computing (independent tasks) or for *workflows* (dependent tasks) which are characterized by parallel coarse grained parallelism [10, 16]. The dependencies of the tasks in a grid workflow are known in advance and can also be modelled by a *scheduling* task graph. Note however, that the task graph used for scheduling is slightly different from the one that is used to describe the concrete execution of the workflow. The scheduling task graph contains additional information about the target environment in which the workflow should be executed. This environmental information is used by the scheduler to find a good assignment of the tasks to the compute nodes and to determine appropriate starting times for these tasks. On the other hand, the *trace* task graph is just another representation of the workflow's execution on the assigned grid resources which is needed by our tuning tool. Before a workflow schedule can be computed a target environment must be established. Scheduling is only possible if a certain Quality of Service (QoS) of the target environment is assured. Thus, an advance reservation (AR) facility is needed to configure an execution environment for a workflow [2, 13, 14]. The execution environment comprises not only the compute nodes but also the network connections for exchanging data. Without AR, one could not plan the execution times of the subtasks because the compute power of unreserved grid nodes could change at any time. Likewise, also the data transfer times could not be considered during scheduling without AR. When the target environment has been established a schedule can be computed and the trace task graph will be derived. It is used by our tuning tool to identify the most critical task, which can then be accelerated by means of code optimization as already explained in the case of MPI applications. Another option could be to reassign the identified task to another compute node or to change the reserved network QoS features. After changing the properties of the target environment by means of AR we could repeat the steps described in this paragraph in order to gradually improve the performance of the work-

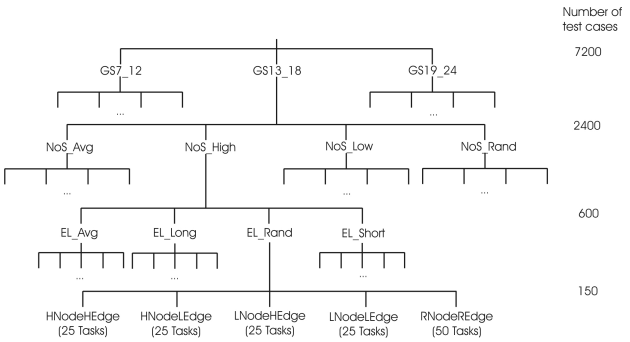

Figure 1: Structure of the OptSched testbench

flow's execution on the grid.

3. Evaluation and Results

For our evaluation we use the OptSched testbench [6] of synthetic task graphs scheduled onto several machine configurations with various static scheduling heuristics.

3.1. Testbench

It is usually difficult to compare the quality of different scheduling algorithms from the literature because each author uses a different set of task graphs for evaluation. Thus, the OptSched testbench has been developed to provide a tool for an objective comparison between scheduling algorithms for homogeneous parallel machines like cluster computers or a collection of grid resources with a certain quality of service. Beyond a large number of scheduling heuristics, the test bench also provides optimal schedule lengths for almost all test cases to allow for absolute comparison. A detailed description of the test bench can be found in [6]. Here, we restrict ourselves to its structure and how it was applied to analyze the proposed tuning tool.

The OptSched test bench is a synthetically created collection of test cases. It allows to select subsets of test cases characterized by specific graph properties. Thus, the users are enabled to investigate the dependance of scheduling algorithms on those properties. OptSched consists of randomly generated task graphs whose size, meshing degree, edge lengths as well as node and edge weights are varied, see **Fig. 1**.

The task graph sizes are grouped into three categories with 7 to 12, 13 to 18 and 19 to 24 tasks per graph, respectively. Regarding the meshing degree, four categories were considered: NoS_Low, NoS_Avg, NoS_High, and NoS_Rand, denoting small, medium and large numbers of dependent tasks created with normal distribution and uniformly distributed numbers of dependent tasks, respectively. The edge-length related EL-categories are organized in a similar manner. The last level of categories is concerned with the ratio between the node weights (computational load) and the edge weights (communication load between processors). E.g. for the HNode-LEdge category we get high node weights and low edge weights, which corresponds to a coarse grained application.

The hierarchically organized testbench comprises a total of 7200 (from bottom to top $150 \cdot 4 \cdot 4 \cdot 3$) task graphs that were considered to be scheduled on five different architectures consisting of 2,4,8,16 and 32 (homogeneous) processing elements, respectively. Thus, the OptSched testbench contains a total of 36000 test cases. All test cases have been scheduled by eight popular static scheduling heuristics, and for almost all test cases an optimal static schedule has been computed in a multi-year effort since 2003. The testbench can be accessed via [5].

3.2. Experiments

For each test case[1] and each available schedule, our tool imports the task graph and the schedule, and represents the mapping of tasks onto processors and the order of tasks on each processor by adding edges with zero weight. We first average over all task graphs and only consider differences between the different scheduling heuristics. Then we consider the influence of the task graph properties on the recommendations.

In the test cases, the average critical path length was between 6 and 7 nodes (ignoring the artificial zero weight source and sink nodes), with a standard deviation of about 2. The task weights comprised from 80 to 87% of the critical path length on average depending on the scheduler, with the optimal schedule showing highest percentage as expected. This means that our tool can achieve an improvement of 12 to 13% for a maximum possible improvement of $t_m = 15\%$.

We performed experiments with $t_m = 10\%, 15\%, 20\%, 25\%$. The results are depicted in **Fig. 2**. We see that we achieve a gain that is increasing with increasing t_m, so the restrictions from outside edges are not so strict that they render attempts at improvement of the critical path useless. Also, while the possible improvement in the case of an optimal schedule is smaller than with heuristic schedules, the difference is small, so that our tool is not only able to improve bad schedules, but is also able to recommend inspection for improvements in the case of optimal schedules. We see these preliminary findings as an indication that our tool is useful.

For $t_m = 15\%$ we have also investigated whether the constraints from outside edges play a role. On average, from 81% to 89% of the tasks recommended for improvement could be improved by t_m, with the optimal schedule providing the smallest value, as expected, and the ETF scheduling heuristic providing the largest value. Thus, the better a schedule is, the larger the influence of the outside edges is. This is no surprise however, because a better schedule means that tasks are packed denser on the pro-

[1] We restricted to task graphs with at most 18 tasks because optimal schedules are available for all of them, and to machine configurations with 2 and 4 processors, because even the optimal scheduler does not use more processors for such small graphs.

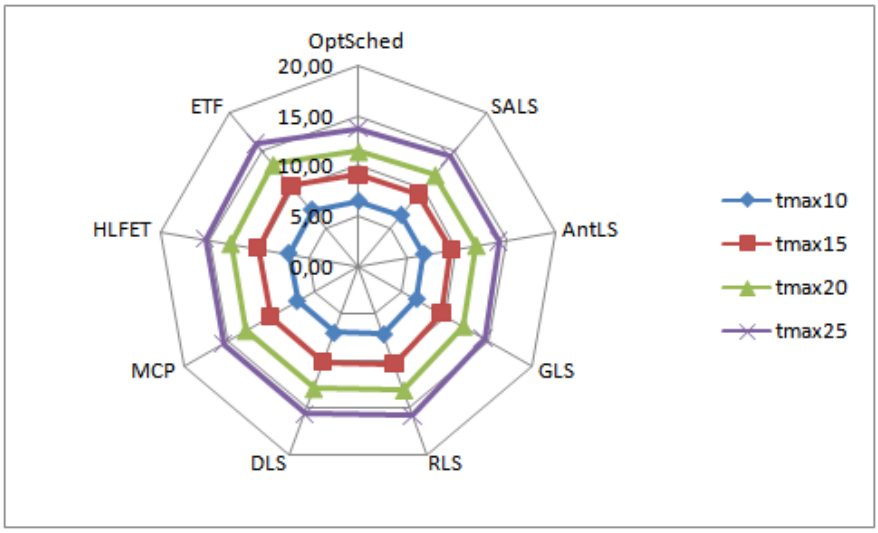

Figure 2: Possible makespan improvements for different scheduling heuristics.

Table 1: Possible makespan improvements for different task graph properties

Property	low	med.	high	rand.
Meshing degree	6.94%	9.25%	10.82%	9.07%
Edge length	11.89%	8.86%	5.88%	9.56%
	l/l	l/h	h/l	h/h
Node/edge weights	8.96%	9.23%	9.03%	8.70%

cessors. Yet, even for weak schedules 11% of the nodes can only be improved to a limit set by an outside edge, so that their influence cannot be ignored.

When considering the task graph properties, we restrict to the optimal schedules for the task graphs, in order to exclude influences from imperfect scheduling, and set $t_m = 15\%$. The results are depicted in **Table 1**. The average possible improvement is largest for a high meshing degree, which may seem surprising as additional edges impose further constraints. However, a low meshing degree typically leads to a short critical path with fewer options for improving tasks. With rising edge length, edges tend to span more than one level of the task graph when ordered topologically, and hence lead to more constraints. Thus, the possible improvement is highest for small edge length. The ratio between node and edge weights has a much smaller influence than the other parameters, while still low node weights and high edge weights lead to largest possible improvements, as one would expect.

4. Extensions

So far we have assumed that all tasks in the task graph are different, i.e. result from different pieces of code, and that the critical path is unique. However, both assumptions often will not hold in practice. For example, if the code contains a function called on two processors, then there will be two tasks resulting from the same piece of code. If one of these tasks is on the critical path and will be recommended for code improvement, then the other task will profit from this improvement as well. This situation can be handled by introducing task types. As SCALASCA logs entry and exit of functions, which is already used to establish a relation between source code and tasks, tasks can be typed on the basis of their associated source code. For each task type present on the critical path, all other instances of this type outside the critical path are not assumed to have a constant runtime but a runtime that improves by a similar factor as the runtime of the task on the critical path does. This has two effects. First, some outside edges in our line graph may not have constant weight anymore, if they contain tasks with a type that is present in a task on the critical path. Second, as an outside edge represents a maximum weight, we cannot compute this maximum numerically anymore. Yet, in the linear optimization, a maximum of two expressions on the outside edge weight simply leads to two inequations that must be fulfilled instead of one.

If there are two critical paths, then we may either start the optimization twice, once for each of them, and recommend the tasks on the path with the smaller improvement. Alternatively, we may couple the two optimization problems, and provide an overall solution. Note that the optimization in the case of a unique critical path may also lead to this situation. If we improve one task so much that an outside edge has the same weight as the corresponding piece of the critical path, then in fact our optimization has created an optimized situation where some part of the critical path is not unique anymore. We now also may employ our optimization procedure iteratively: first on the critical path, then on the two partial critical paths, and so on. Thus, we can trade runtime of the optimizer against additional possibilities for code improvement.

5. Conclusions

We have presented an extension of a tool for guiding performance improvements in grid applications, to make the tool accessible to both grid workflows and MPI applications, i.e. with different levels of granularity, and to both task graphs with task runtimes known in advance, and task graphs derived from traces of a run. The extensions comprise the typing of tasks so that improving one task of a certain type will also improve the runtime of all other instances of this task type; also we incorporated the case of a non-unique critical path, which allows to iteratively refine the process of improvement.

We have furthermore evaluated the tool with a benchmark suite of synthetic task graphs and schedules. The results are promising in the sense that on average, a reasonable potential for make span improvement by code acceleration could be found.

Future work will first concentrate on applying our tool on real world applications, probably starting with the Tachyon application from the SPEC MPI2007 benchmark suite.

Beyond, there are two directions to follow. First, so far we did not model the effort to improve a code function by a certain percentage, but assume it to be constant for all tasks. By correlating function metrics from software engineering (e.g. lines of code) with expected effort to improve code, we might provide the total amount of necessary programming work to carry out the improvements and maximize the expected gain in runtime for this given amount.

Second, one may also invest the suitability of our tool to data-parallel applications. In that case, a task graph with source at the top and sink at the bottom will look like a number of vertical paths, representing the computation on the processors, with some horizontal edges indicating synchronization or all-to-all communications or the like. Such a situation can be recognized by our tool in the form of a number of identical critical paths. If it does, it may recommend to even out waiting times, which are known but not considered so far, instead of reducing task runtime.

Acknowledgements

We thank Felix Wolf and Bernd Mohr for fruitful and helpful discussions.

References

[1] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Computing*, to appear 2009.

[2] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proc. International Workshop on Quality of Service*, 1999.

[3] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM/MPI Conference*, pages 303–312, 2006.

[4] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, July 2009.

[5] Udo Hönig. Optimal schedules homepage, 2009. http://valexia.fernuni-hagen.de/OptSchedHome.html.

[6] Udo Hönig and Wolfram Schiffmann. A comprehensive test bench for the evaluation of scheduling heuristics. In *Proc. 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'04)*, pages 437–442, 2004.

[7] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[8] Jörg Keller and Wolfram Schiffmann. Guiding performance tuning for grid schedules. In *Proc. 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, May 2009.

[9] M. Matsuda, Y. Ishikawa, Y. Kaneo, M. Edamoto, F. Okazaki, H. Koie, R. Takano, T. Kudoh, and Y. Kodama. Overview of the GridMPI version 1.0 (in japanese). In *Proc. SWoPP*, 2005.

[10] Aline P. Nascimento, Cristina Boeres, and Vinod E. F. Rebello. Dynamic self-scheduling for parallel applications with task dependencies. In *Proc. 6th International Workshop on Middleware for Grid Computing (MGC '08)*, pages 1–6, New York, NY, USA, 2008. ACM.

[11] Aline P. Nascimento, Alexandre C. Sena, Cristina Boeres, and Vinod E. F. Rebello. Distributed and dynamic self-scheduling of parallel mpi grid applications: Research articles. *Concurr. Comput.: Pract. Exper.*, 19(14):1955–1974, 2007.

[12] Martin Schulz. Extracting critical path graphs from MPI applications. In *Proc. IEEE International Conference on Cluster Computing (Cluster 2005)*, pages 1–10, 2005.

[13] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *Proc. 14th International Parallel and Distributed Processing Symposium*, 2000.

[14] Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, pages 137–153, 2000.

[15] John Stone. Tachyon parallel / multiprocessor ray tracing system website, May 2009. http://jedi.ks.uiuc.edu/ johns/raytracer/.

[16] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, September 2005.