VizzScheduler - A Framework for the Visualization of Scheduling Algorithms

Welf Löwe and Alex Liebrich

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe
Postfach 6980, 76128 Karlsruhe, Germany
E-mail: {loewe|liebrich}@ipd.info.uni-karlsruhe.de

Abstract. The computation of efficient schedules of task graphs for parallel machines is a major issue in parallel computing. Such algorithms are often hard to understand and hard to evaluate. We present a framework for the visualization of scheduling algorithms. Using the *LogP* cost model for parallel machines, we simulate the effects of scheduling algorithms for specific target machines and task graphs before performing time and resource consumptive measurements in the real world.

1 Introduction

Parallel programs can be modeled by task graphs defining a set of parallel tasks and control and data dependencies between them. A schedule of a task graph defines the mapping of its tasks to parallel target machines. Finding an optimal mapping is an *NP* hard problem for many task graph classes and many models for parallel architectures.

A great variety of scheduling algorithms for different task graph classes and cost models has been published. We also contributed some algorithms and experienced that it is hard to evaluate their performance in practice. The performance depends (among others) on (1) the ratio for computational costs of tasks and communication costs on a specific architecture (granularity of the problem), (2) the structure of the dependencies between tasks, (3) the ratio of tasks that may potentially run in parallel and the actual number of processor in a specific parallel machine. Measurements evaluating the schedules over a variety of values of (1) – (3) are extremely time and resource consumptive. Often it is hard to access more then one parallel machine and, hence, such experimental results are rather rare. On the other hand, we used to be quite disappointed when we took the effort to evaluate some theoretical promising algorithm and found it poor in its practical performance. We therefore decided to design a framework allowing a rapid evaluation of scheduling algorithms. We restrict the experimental evaluations to only those scheduling algorithms that passed successfully our simulations.

Another problem with scheduling algorithms addresses their understanding. We found scheduling algorithms published imprecise, misunderstanding of even wrong. As the feed back showed, our own publications were no exceptions. It often requires some attempts to finally find the intended implementation. Therefore, the framework allows a step-by-step execution visualizing together the task being scheduled, the schedule (as computed so far) and the current line of program code in the scheduling algorithm. This execution mode is a good tool for testing and debugging the algorithms before applying or publishing them. As a by-product, it turned out that communicating new scheduling algorithms in presentations became much easier when using our VizzScheduler.

The body of the paper is organized as follows: Section 2 discusses some ideas providing a basis for our framework. Section 3 shows the general architecture of the framework. It gives an idea how simple it is to integrate new task graphs and new scheduling algorithms. It demonstrates the work of the framework on an example. Finally, Section 4 concludes the paper.

2 Basics

To abstract from the diversity of parallel machines, we choose an abstract *machine model*. Culler et al introduced the LogP model [1]. It assumes a finite number of processors with local memory linked by a network connection. It abstracts from the network topology, presuming that the position of the processor in the network has no effect on the communication costs. Each processor has its own clock, synchronization and communication is done via message passing. To evaluate schedules, a cost model is required reflecting the latency for point-to-point communication in the network, the overhead of communication on processors themselves, and the network bandwidth. The *LogP* machine [1] models these communication costs with parameters ***L****atency*, ***o****verhead*, and ***g****ap* (which is actually the inverse of the bandwidth per processor). In addition to *L*, *o*, and *g*, a parameter *P* describes the number of ***P****rocessors*. These parameters have been determined for a number of parallel machines including the CM-5 [1], the IBM SP1 machine [3], a network of workstations and a powerXplorer [2]. In addition to the machine parameters, we need the input size of the problem and the runtime of the single tasks to predict the runtime of the scheduled program. The latter is usually obtained from measurements of the sequential program or a simple scheduled parallel program. It clear that this leads to errors due to different caching effect and tasks with input dependent load cannot be captured at all. However, using this approach leads to precise predictions of parallel programs at least for a large class of problems. In practice, the runtime measurements of parallel program confirmed their *LogP* based predictions. Parallel programs can be modeled by *task graphs*. A task graph is a directed acyclic graph; nodes correspond to computations, edges to communications between them. A *LogP schedule* is a set of sequences of computations, send, and receive operations and their starting times corresponding to the tasks and edges of the task-graph. Parameter *P* restricts the cardinality of the set. The starting times must conform the restrictions imposed by the task graph and the *LogP* parameters. There are quite a few *LogP* scheduling algorithms published, e.g. in [1,2,5,6].

In order to evaluate scheduling algorithms, we only need the *LogP* runtime prediction for a given problem of a given size. To debug or explain the algorithms, we additionally need an architecture allowing to load a scheduling algorithm, execute it step-by-step and draw temporary results. Therefore, this architecture needs access the relevant internal data structures of the scheduling algorithm in each step. The *Java Debug Architecture* [4] provides such an architecture for Java programs. It allows to launch a runtime environment for a program, to start a debugee (scheduling algorithm) and to control the execution of this debugee by another program. Additionally, the control program can access the state of the debugee in each execution step. We therefore decided to implement the framework and the scheduling algorithms in Java.

The two main data structures that should be visualized are the task graph and its schedule graph generated by the scheduling algorithm. The former is a directed, acyclic graph; its node represent computational tasks its edges essential dependencies between them. The latter is usually visualized by Gantt charts, a graph showing the computations of the single processors over time. Gantt charts are quite easy to handle since the fix coordinates system of processors and time reduce the drawing problem to a simple scaling. Actually, we could try to find a permutation of the processors minimizing the edge crossings. As the layout is computed online while the scheduling algorithm runs, this approach would lead to animations that potentially change the order of processors whenever an edge is inserted. Tasks graphs, or directed acyclic graphs (*DAG*s) in general, do not have such a canonical drawing. An optimal drawing also minimizes edge crossings, its efficient computation is not possible, unless *P=NP*. Our framework does not try to innovate *graph drawing*. Good approximation algorithms are known, e.g. heuristic spring embedding approaches, Sugiyama's algorithm [7] (preserving the layered structure of *DAG*s, i.e. nodes with the same depth are drawn on the same *y*-coordinate) or so called upward drawings. These approaches are sufficient for our purpose and used in the framework.

3 The Architecture

We developed a more general framework – the VizzEditor – that supports the rapid design of visualizations of general algorithms. The tool presented here – the VizzScheduler – is an instance of the VizzEditor.

The VizzEditor accesses the Java Debug Interface and provides certain imaging tools. It enables the integration of user defined graphical tools viewing aspects of the programs that are chosen by the user. In addition to some static information the VizzEditor can draw dynamic program information with graphical views. Therefore, the information is read on certain user-defined breakpoints and mapped to drawing operations of instances of the graphical views. A general controller coordinates the synchronous update of the program to visualize and the graphical views. Each breakpoint hit generates an event object sent to the controller. This object holds information about the breakpoint (necessary for some filtering) and entries to access the program state (necessary to obtain the runtime data). It passes this object to the mapping program which in turn performs filtering and calls the drawing operations. In many cases the predefined graphical tools are sufficient. To visualize a new algorithm, the user implements a new mapping program, a simple Java class, and determines the breakpoint in the implementation of the algorithm to animate.

We can even reduce these user tasks to a minimum if we know the class of algorithms to be animated in advance. Then we write an abstract class that contains all the problem specific data and operations for this class of problems. The new algorithm extends this abstract class and, thus, runs in the predefined environment. If the breakpoints are placed in the predefined code of the abstract class, it is possible to define a general mapping program for all algorithms of this class in advance. We applied this approach to the class of scheduling algorithms leading to the VizzScheduler. It restricts the algorithms to visualize to *simulations of parallel programs* and *scheduling algorithms* for task graphs. Simulations are visualized by task graph constructions. A step-by-step execution of a parallel program (simulation) shows the task graph construction; it visualizes program points corresponding to computations and communications. The scheduling algorithms take arbitrary task graphs (as generated by a parallel program simulation) and generate a mapping to an instance of the *LogP* machine. Scheduling algorithms are visualized by Gantt charts. A step-by-step execution shows the decision of the algorithm to place a task on a certain processor starting at a certain time.

We implemented 4 major classes representing our application context:

1. one class models the *task graphs* as generated by simulations of parallel programs.
2. an abstract class modeling this class of the *simulations of parallel programs*. It captures the input and input size of the algorithm. Each simulation constructs an instance of the task graph class. This instance is passed to the subclasses of
3. an abstract class modeling the *scheduling algorithms*. It accesses the *LogP* parameters (captured in an auxiliary class) and the task graph to be scheduled. Each scheduling algorithm must construct an instance of
4. a class modeling the *schedules* themselves. This class contains the constructors for correct *LogP* schedules and captures the schedule information.

The breakpoints are set on the constructor methods of the task graph and schedule classes. This is sufficient to animate the task graph and schedule construction algorithms, respectively. Therefore, we predefined also the two programs (classes) mapping the breakpoint events to drawings.

The user of the VizzScheduler looks for a simulation of a parallel program in the library or implements, if necessary, a new simulation class (inheriting form the corresponding abstract class that is predefined in the framework, cf. 2.). Such a simulation calls the constructor methods in the class modeling the task graph (cf. 1.). Running a simulation automatically triggers the drawing of the corresponding task graph. Then the user looks for a scheduling algorithm in the library or implements a new scheduling algorithm class (inheriting form the corresponding abstract one, cf. 3.). The schedule construction uses the construction methods provided by the schedule class (cf. 4.) Each run automatically triggers the drawing of the corresponding Gantt chart. In the VizzScheduler GUI, the user may vary the LogP parameters, the input size and the computation times of the task graph nodes.

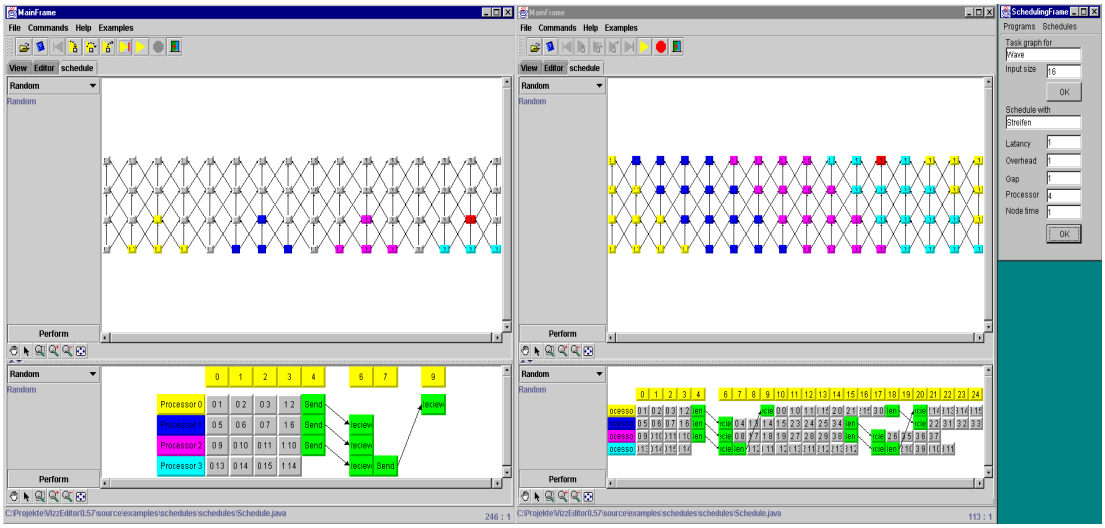

Fig. 1. A scheduling algorithm for the one-dimensional wave simulation. Left frame shows the algorithm in action, the right frame the final schedule. The single frame top, right allows the change of the *LogP* parameters.

Example 1: Fig. 1 displays a scheduling for an algorithm implementing a one-dimensional wave simulation. The final schedule maps diagonal stripes of the task graph to the same processor. Actually, it maps triangle shapes to the same processor, inserts necessary communications, and then proceeds in scheduling the tasks. This approach is hard to explain on the final schedule but easy to show in the process of scheduling.

4 Conclusions

We introduced the VizzScheduler framework allowing the rapid design scheduling algorithm animations. Designers of such algorithms can evaluate their performance for a range of task graphs and machines and pre-select candidates for evaluation in time and resource consumptive experiments. Additionally, the framework helps to debug or present such algorithms. Get a free version from

`http://i44pc29.info.uni-karlsruhe.de/VizzWeb`

References

1. D. Culler, R. Karp, and D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 93)*, pp 235-261, 1993.
2. J. Eisenbiegler, W. Löwe, W. Zimmermann. Optimizing parallel programs on machines with expensive communication. In *Europar' 96 Parallel Processing II*, LNCS 1124, pp. 602-610. Springer, 1996.
3. B. Di Martino and G. Ianello. Parallelization of non-simultaneous iterative methods for systems of linear equations. In *Parallel Processing: CONPAR 94 - VAPP VI*, LNCS 854, pp. 253-264. Springer, 1994.
4. Java Debug Architecture: http://java.sun.com/j2se/1.3/docs/guide/jpda/index.html.
5. W. Löwe and W. Zimmermann. Scheduling balanced task graphs to LogP-Machines. *Parallel Computing* 26, pp. 1083-1108, 2000.
6. R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 142-153. ACM, 1993.
7. K. Sugiyama, S. Tagawa, and M. Toda: Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst. Man., Cybern,, SMC-11*, pp. 109-125, 1981.