PEELSCHED: a Simple and Parallel Scheduling Algorithm for Static Taskgraphs

Jörg Keller, Rainer Gerhards

Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
joerg.keller|rainer.gerhards@fernuni-hagen.de

Abstract: We present a new algorithm, which we call PEELSCHED, to schedule a set of tasks with precedence constraints and communication costs onto a parallel computer with homogeneous processing units. The algorithm is deterministic and simple, and can be parallelized itself. The new algorithm is based on the usage of structural graph properties, in particular series-parallel or N-free graphs, but can be used on any DAG. We provide an implementation and validate it against a benchmark suite of task graphs. We find that the algorithm's scheduling results are comparable to strictly sequential schedulers.

1 Introduction

Scheduling of problems given by static task graphs onto parallel machines is a problem that has received continuous attention since half a century. Already in 1966, Graham published the famous list scheduling algorithm [Gra66]. As the problem of achieving optimal makespan is NP-hard, a wealth of scheduling heuristics has been developed since then. While some of them achieve better makespan by increased scheduling time, the majority of scheduling algorithms is sequential in nature. A notable exception is the Parallel Bubble Scheduling and Allocation (PBSA) algorithm [Kwo97].

We present a simple, deterministic scheduling algorithm called PEELSCHED with the following goals: not too complicated, parallelizable, and comparable in result to other deterministic schedulers like Dynamic Level Scheduling (DLS) [SL93] or Highest Level First with Estimated Times (HLFET) [ACD74]. To achieve these goals, we first consider scheduling for so-called series-parallel or N-free graphs, and then extend our observations to arbitrary task graphs given by DAGs (directed acyclic graphs). We finally derive our simple and greedy algorithm.

We evaluate the scheduling algorithm with the help of the OptSched benchmark suite [Hön09, HS04] of synthetic task graphs. This suite is large, and besides schedules for a multitude of heuristics also optimal schedules are available for most task graphs. Therefore, both a relative and an absolute evaluation is possible, see [Hön08]. Furthermore, the task graphs in the suite are classified according to several parameters like number of

edges, edge length, choice of node weights and edge weights, so that also a more detailed evaluation is possible. The result of this evaluation is that the PEELSCHED algorithm leads to makespans comparable to other deterministic scheduling heuristics, i.e. which on average are about 8% longer than corresponding optimal schedules. Also, even on small task graphs, the amount of parallelism is between 2 and 3, so that a quad-core processor can be employed to run the scheduler in parallel.

The remainder of this article is organized as follows. In Sect. 2 we briefly summarize static scheduling of task graphs and series-parallel task graphs. In Sect. 3, we describe the PEELSCHED algorithm and the rationale behind its design, and in Sect. 4 we present experimental results. In Sect. 5 we conclude and give an outlook to future research.

2 Series-parallel Taskgraphs

We employ the usual model of static scheduling: an application is given as a set V of n tasks, where each task i has a runtime r_i. Each task consumes input only at the start (denoted by incoming edges), and produces output only when it terminates (denoted by outgoing edges). Each edge (i, j) is assigned a communication time $c_{i,j}$ that reflects the amount of data to be transported from task i to task j. Tasks and edges together comprise the appliation's *task graph*.

The machine to execute the application consists of p identical processing units interconnected by a network. At any point in time, each processing unit executes at most one task, and a task started on a processing unit runs to completion without interruption. If tasks i and j connected by edge (i, j) run on the same processing unit, then the communication time $c_{i,j}$ is ignored because communicating data via main memory is assumed to be much faster than via the network.

A *schedule* is a mapping $m : V \to \{1 \dots p\}$ of the tasks onto the processing units, so that furthermore each task is assigned a start time s_i and an end time e_i. A schedule is *valid* if

1. for each task i, $e_i = s_i + r_i$,
2. for all tasks i, j mapped onto a particular processing unit, either $e_i \leq s_j$ or $e_j \leq s_i$,
3. for each edge (i, j), if tasks i and j are mapped to different processing units, then $s_j \geq e_i + c_{i,j}$, and $s_j \geq e_i$ otherwise.

We typically assume $\min_{i \in V} s_i = 0$, i.e. the schedule starts at time 0. Then $\max_{i \in V} e_i$ is called the *makespan* of the schedule, i.e. the time when the last processing unit completes its last assigned task.

The fact that the precedence constraints must be considered during scheduling renders parallelization of scheduling algorithms complicated. The PBSA-algorithm [Kwo97] partitions the task graph along the critical path in P parts, if the scheduler runs on a P-processor machine, each processor schedules one part separately, and the resulting partial schedules are merged and violations removed to obtain a valid schedule.

A simplification would be achieved if the task graph itself allows a division of the scheduling work. This holds true for the class of *series-parallel* task graphs.

A task graph $G = (V, E)$ is series-parallel if it consists either of a single node, or if it can be split into two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $V = V_1 \cup V_2$ such that one of the following conditions holds true

S-step: V_1 contains all sources of G, V_2 contains all sinks of G, and the sinks of G_1 and the sources of G_2 form a complete bipartite graph in G; i.e. G can be expressed as the serial composition of G_1 and G_2;

P-step: G_1 and G_2 form different (weakly) connected components of G; i.e. G can be expressed as the parallel composition of G_1 and G_2;

and both G_1 and G_2 are series-parallel.

Figure 1 depicts an example graph together with the partitioning according to S-steps and P-steps that demonstrate that this graph is series-parallel. Note that the addition of a single edge, e.g. $(2, 5)$, can destroy the series-parallel property, as the sub-graph consisting of nodes 1 to 6 could not be partitioned anymore.

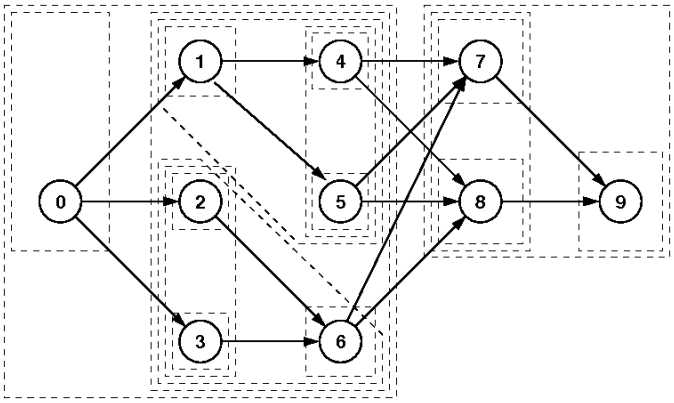

Figure 1: Example of series-parallel graph with recursive partitioning. Presence of edge $(2, 5)$ would destroy the series-parallel structure.

Series-parallel graphs are also denoted N-free graphs, as the 4-node graph with N-shape depicted in Fig. 2 is the smallest structure that is not series-parallel. Thus, if a graph comprises such an N-structure (where (a, d) may well be a path instead of an edge), it is not series-parallel, or put vice versa, if it is series-parallel, it cannot contain such an N-structure, i.e. it is N-free.

We note that to transform the N-structure into a series-parallel graph, one must either remove the diagonal edge (a, d), or must add a second diagonal edge (b, c) to achieve a complete bipartite graph between $\{a, c\}$ and $\{b, d\}$.

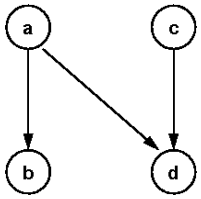

Figure 2: Smallest non-series-parallel graph in N-shape. To transform into a series-parallel graph, either edge (c, b) must be added or (a, d) removed.

Scheduling a series-parallel graph can be parallelized, although the resulting schedule will not necessarily be optimal, as scheduling is NP-hard even for the class of series-parallel graphs, and even if we allow an arbitrary number of processing units and restrict the possible communication costs to 0 and 1 [MS99].

There exist linear-time algorithms to compute the hierarchical partitioning of a series-parallel graph, see e.g. [VTL82]. Once this partitioning is known, a scheduling algorithm for series-parallel task graphs can proceed as follows.

Let us assume that we know the partitioning of a task graph G in sub-graphs G_1 and G_2. If the partitioning was done by an S-step, then we recursively schedule G_1 and G_2, and combine their schedules by concatenating the schedule for G_2 to the schedule of G_1. By concatenating we mean adding makespan$(G_1) + \max c_{i,j}$ to all start and end times in the schedule for G_2, where i is an arbitrary sink of G_1 and j is an arbitrary source of G_2. The resulting makespan is the sum of the two makespans plus the maximum communication time of the edges connecting the two sub-graphs.

If the partitioning was done by a P-step we can simply proceed as with the S-step above. This is sufficient if both G_1 and G_2 are large enough to take advantage of all of the p processing units. Thus, although the two sub-graphs are independent and could be executed in parallel, they are serialized because each can make use of the complete machine. Alternatively, one could partition the p processing units into two sets of p_1 and $p_2 = p - p_1$ processing units each, and recursively schedule G_1 onto p_1 and G_2 onto p_2 processing units, respectively. Finding a good value of p_1 is not obvious, but using

$$p_1 = p \cdot \frac{\text{sum of runtimes in } G_1}{\text{sum of runtimes in } G}$$

should be a good estimate. Now, the two sub-graphs are executed in parallel, and the resulting schedule is simply the union of the two partial schedules. The resulting makespan is the maximum of the two sub-graphs' makespans.

As the two recursive calls in both the S-step and the P-step are independent, they may be executed concurrently, so that the scheduling algorithm itself is a parallel program.

3 The PEELSCHED Algorithm

So far, we have seen a generic procedure to schedule series-parallel task graphs. To apply this procedure to arbitrary task graphs, it is necessary either to extend the procedure or to transform the task graphs. When one pursues the latter approach, one first has to find all N-structures in a task graph, and then remove them. As explained above, this can be done either *conservatively* by adding a second diagonal edge with communication weight 0 (even $-\infty$ would be possible), or *optimistically* by removing the diagonal edge.

Figure 3 depicts how the example graph of Fig. 1, enriched with the edge $(2, 5)$, can be transformed in both ways. We note that also in the optimistic approach, we do not result in the original graph of Fig. 1. This exemplifies that the optimistic approach may remove more edges than necessary.

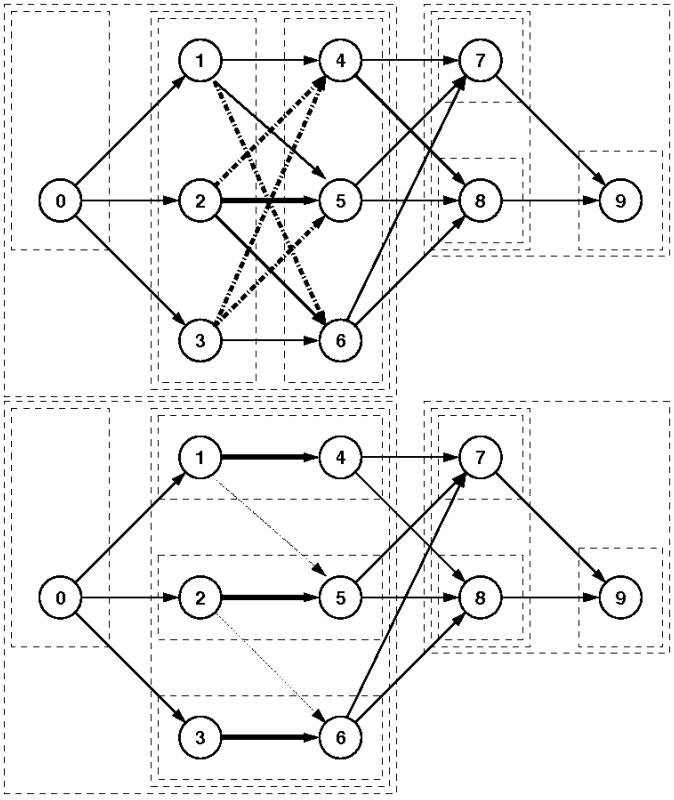

Figure 3: Transformation of counter example graph by addition (upper figure, dashed edges) or removal (lower figure, dotted edges) of edges.

The terms conservative and optimistic are chosen similarly to their use in parallel discrete event simulation:

- In the conservative transformation, the schedule for the transformed graph is also valid for the original graph, as all precedence constraints have been considered, and only additional edges, i.e. constraints, have been introduced in the transformed graph. However, the additional, artificial constraints may result in a schedule with a longer makespan.
- In the optimistic transformation, the schedule for the transformed graph provides the possibility to achieve a better makespan because constraints have been removed. However, there exists the possibility that this schedule is not a valid schedule in the original graph, because requirement 3 is violated for a removed edge. In this case, the violation must be repaired by increasing the start time of task j appropriately, which in turn may necessitate to delay other tasks mapped to the same processing unit with a start time larger than that of j, to avoid a violation of requirement 2. Also, this may result in another violation of requirement 3, where task j now takes the place of previous task i.

The conservative variant has been used in [Mit04] when visualizing arbitrary graphs with a routine targeted for series-parallel graphs.

We apply the optimistic variant, but try to avoid the explicit computation of all N-structures, and the corrections after violation of schedule requirements. We achieve this by employing a greedy approach for S-steps, and use of Graham's List scheduler.

The greedy approach is based on the following observation. Often, a task graph has only a single source node. In this case, an S-step is always possible by using the single source as sub-graph G_1, and the rest of the task graph as G_2. Also, a violation cannot occur here. Hence, our approach first removes *all* source nodes of the task graph and uses them as G_1 in a kind of pseudo-S-step. If there is more than one source node, and the source nodes are not connected to G_2 by a complete bipartite graph, a violation may occur, which must be corrected afterwards.

To increase the possibility of a P-step on the rest of the task graph, we also remove the sink nodes of the task graph, because the frequent case of a single sink node would prevent the rest of the graph to fall into several weakly connected components.

Then we recursively call our algorithm on each weakly connected component of the remaining graph.

As we repeatedly "peel" the source and sink nodes from the task graph like we peel an apple, we call our algorithm PEELSCHED. Figure 4 depicts the pseudo-code of the algorithm.

So far, we did not care to construct a schedule, but simply generate a linear order of the nodes by concatenating the list of source nodes, the lists of nodes from the recursive calls, and the list of the sink nodes. To generate a schedule, we use Graham's List scheduler [Gra66] as a mapping algorithm. The List scheduler, originally designed for tasks without precedence constraints, maps the tasks one by one in list order. The next task is mapped to the processing unit that is available at the earliest time, and thus minimizes the makespan of the resulting partial schedule. For task graphs with precedence constraints, those constraints can be taken into account when mapping the next task. This is a standard approach

```
List peelsched(Graph G){
int num, i;
Graph H[MaxCC];
List sourcelist, sinklist, complist;

if(size(G)==0) return NIL; // empty graph,
recursion end

sourcelist = removesources(G); // Pseudo-S-step
sinklist = removesinks(G);
num = conncomponents(G,H[]); // P-step
for i=0 to num-1 do in parallel{
    complist = peelsched(H[i]);
    sourcelist=concat(sourcelist,complist);
}
return concat(sourcelist,sinklist);
}
```

Figure 4: Pseudo-code of PEELSCHED

in static task scheduling, and in this way, violations of requirement 3 are handled automatically.

Alternatively, to avoid the sequential List algorithm after the parallel PEELSCHED algorithm, one can employ the List scheduler directly on source and sink lists, which is easy because source nodes do not have edges among themselves. The same holds for sink nodes. Thus, the List scheduler can be employed as originally conceived: for tasks without precedence constraints. Then, instead of concatenating the lists as depicted in the pseudo-code, one concatenates the partial schedules from sources, recursive calls, and sinks. As the schedules from multiple recursive calls are independent, they might be overlapped in a post-processing step to increase efficiency and reduce makespan, as proposed in [HS06].

Note that the majority of the runtime currently is spent in the computation of the connected components. Several possibilities for improvement exist. In the first stages of the recursion, where the graphs are still large, and only one or few threads are busy, a parallel algorithm for connected components can be employed to utilize the idle threads. Also, instead of computing connected components from scratch in each recursion step, one could employ a dynamic connected components algorithm, that only tests whether the removal of the edges adjacent to sources and sinks has split the component. Moreover, when the graph G has reached a size below a certain threshold, one might employ a direct scheduling algorithm such as HLFET.

4 Experimental Results

We have implemented a sequential and a parallel version of the PEELSCHED algorithm to test its ability to achieve makespans comparable to other deterministic heuristics. We have

generated schedules for the task graphs from the OptSched benchmark suite. Furthermore, we have generated some larger graphs with the same generator and the same parameters as used for OptSched, to get an impression as to its scalability.

With respect to makespan, we find that the optimal schedule is found in about one third of all cases. Otherwise, the schedules found are on average 9% longer than the optimum. The maximum deviation from an optimal schedule was 29% longer, for a machine with 2 processors. This overall result roughly corresponds to figures for algorithms like DLS and HLFET, see [Hön08]. When we classify the task graphs according to the number of processors they are scheduled on, then from 4 processors on, about 36% of the schedules are optimal, with an average additional length of 8% in case of non-optimal schedules. Hence, PEELSCHED is able to find efficient schedules already for small machines sizes. When we classify the task graphs according to their sizes (7 to 12 tasks, 13 to 18 tasks, 19 to 24 tasks), then the fraction of optimal schedules found shrinks from 50% to 25% and 11%, but the average deviation in case of non-optimal schedules also shrinks from 10% to 8%. Hence, PEELSCHED is able to find efficient schedules for task graphs also with growing size. When we classify the task graphs according to edge length (short, medium, long, random), then we find that only for long edges the average deviation from optimal makespan increases from 8% to 10%, which is still not much. Hence, PEELSCHED is not sensitive to edge lengths in the task graphs. When we classify the task graphs according to edge and node weights (both can be high, low, or random), then we find that for high edge weights combined with low node weights, the average deviation from optimal makespan increases to 16%, and the fraction of optimal schedules decreases to 20%. In the opposite situation (low edge weights and high node weights), the average deviation is 5% while almost half of the schedules are optimal. Hence, not surprisingly, task graphs where the edge weights dominate are difficult to schedule for PEELSCHED.

With respect to parallelization, we find that at most 6 threads could be employed. Yet, as we do not employ pseudo P-steps where we remove diagonals in N-structures, this is not really surprising. Also, when parallelizing the connected components algorithm (see the note at the end of the previous section), more threads can be employed.

We have also generated 1,000 graphs with 500 to 1,500 nodes, chosen with parameters (apart from graph size) similar to the graphs from OptSched. As we do not have optimal schedules for these graphs, we compare PEELSCHED with schedules generated by HLFET. Here, the PEELSCHED schedules have a makespan about 2.9% longer than those from HLFET. If we assume that HLFET schedules are on average 8% longer than optimal schedules, then the schedules generated by PEELSCHED are about 11% longer than optimal schedules, which is still very close. For those graphs, at most 31 threads could be used, yet in the majority of the cases not more than 10 threads are used, which indicates that to employ more parallelism, pseudo-P-steps and parallel algorithms are necessary. This could also reduce the runtime, which is currently higher than that of HLFET.

5 Conclusions

We have presented a simple, deterministic, and parallelizable scheduling algorithm for static task graphs, that we call PEELSCHED. We have detailed the rationale behind its design, which comes from observations on scheduling series-parallel task graphs. We have done a test of its ability, i.e. whether it produces schedules with a length close to the optimum makespan, and comparable to that of other sequential deterministic static schedulers. The test was performed on the complete OptSched benchmark suite, and indicates that PEELSCHED is competitive, but will have to use further measures to achieve better parallelism.

So far, our use of the series-parallel paradigm is only implicit. Future work will concentrate on experimenting with an explicit construction of the series-parallel partitioning together with detection of N-structures and application of the conservative or optimistic transformation. In a refinement, the choice of transformation may be adaptive, depending e.g. on the weight of the edge.

Furthermore, a larger class of graphs to be used is the class of N-sparse graphs, that contain at most one N-structure among any 5 nodes of the graph. Transforming task graphs to this class will require fewer insertions and/or deletions of edges, and [SW99] presents an algorithm to derive a linear extension of the task graph nodes, i.e. a linear ordering of the nodes, that reflects the precedence constraints and that minimizes the number of chains. A chain is a sequence of nodes all depending on their predecessors in the sequence, and thus each chain can be completely mapped on one processor, without having to consider the nodes one by one.

Acknowledgements

We thank Winfried Hochstättler for mentioning the "N-free" formulation of series-parallel graphs and the treatment of those graphs in mathematics. We also thank Udo Hönig for sharing his immense knowledge about scheduling algorithms. J. Keller would like to thank Christoph Kessler for an invitation to present this research as a work in progress during a guest lecture, which improved the presentation of the material.

References

[ACD74] T. L. Adam, K. M. Chandy, and J. Dickson. A Comparison of List Scheduling for Parallel Processing Systems. *Comm. ACM*, 17:685–690, 1974.

[Gra66] R. L. Graham. Bounds for certain multi-processing anomalies. *Bell Syst. Tech. Journal*, 45:1563–1581, 1966.

[Hön08] Udo Hönig. *Optimales Task-Graph-Scheduling für homogene und heterogene Zielsysteme*. PhD thesis, FernUniversität in Hagen, Fak. Mathematik und Informatik, 2008.

[Hön09] Udo Hönig. Optimal Schedules Homepage, 2009. http://valexia.fernuni-hagen.de/OptSchedHome.html.

[HS04] 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.

[HS06] Udo Hönig and Wolfram Schiffmann. A Meta-algorithm for Scheduling Multiple DAGs in Homogeneous System Environments. In *Proc. 18th Int.l Conf. Parallel and Distributed Computing and Systems (PDCS)*, 2006.

[Kwo97] Y.-K. Kwok. *High-Performance Algorithms for Compile-Time Scheduling of Parallel Processors*. PhD thesis, University of Hongkong, 1997.

[Mit04] Margaret Mitchell. Creating Minimal Vertex Series Parallel Graphs from Directed Acyclic Graphs. In Neville Churcher and Clare Churcher, editors, *Australasian Symposium on Information Visualisation, (invis.au'04)*, volume 35 of *CRPIT*, pages 133–139, Christchurch, New Zealand, 2004. ACS.

[MS99] Rolf H. Möhring and Markus W. Schäffter. Scheduling series-parallel orders subject to 0/1-communication delays. *Parallel Computing*, 25(1):23–40, 1999.

[SL93] G. C. Sih and E. A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):75–87, 1993.

[SW99] Rainer Schrader and Georg Wambach. The setup polytope of N-sparse posets. *Annals of Operations Research*, 92:125–142, 1999.

[VTL82] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The Recognition of Series Parallel Digraphs. *SIAM Journal on Computing*, 11(2):298–313, 1982.