Fast optimal task graph scheduling by means of an optimized parallel A*-Algorithm

Udo Hönig and Wolfram Schiffmann

FernUniversität Hagen, Lehrgebiet Rechnerarchitektur, 58084 Hagen, Germany
{Udo.Hoenig, Wolfram.Schiffmann}@FernUni-Hagen.de
http://www.informatik.ti1.fernuni-hagen.de/

Abstract. The development of high speed local networks and cheap, but also powerful PCs, lead to an extensive use of PCs as building blocks in modern parallel computer systems. In order to exploit the available resources at the best, any program has to be split into parallel executable tasks, which have to be scheduled to the available processing elements. The need for data communication between these tasks leads to dependencies, which strongly effect the schedule. In this paper, we consider task graphs that take computation and communication costs into account. For a completely meshed homogeneous computing system with a fixed number of processing elements, we compute schedules with minimum schedule length. Our contribution consists of parallelizing an informed search algorithm for calculating optimal schedules based on the IDA*-algorithm, a memory-saving derivative of the well known A*-algorithm. Due to the resulting memory requirements, the application of the A*-algorithm is restricted to task graph scheduling problems with a quite small number of tasks. In contrast, the IDA*-algorithm can compute optimal schedules for more complex task graphs.

1 Introduction

As PCs and computer networks advance, they become attractive building blocks for cheap and powerful parallel computing systems. A parallel program consists of (sub)tasks that can be executed in parallel. In order to exploit effective parallelism these tasks must be assigned to the available processing elements and the starting times on those processing elements must be determined. The dependencies between the tasks of a parallel program can be described by means of a *task graph* that consists of a directed acyclic graph (DAG) [6].

In this paper, the objective of solving the *task graph scheduling problem* is to minimize the overall computing time. The time that a task i needs to compute an output by using the results from preceding tasks corresponds to the working load for the processing element to which that task is assigned. It is denoted by a node weight w_i of the task node. The cost of communication between two tasks i and j is specified as an edge weight c_{ij}. If both tasks are assigned to the *same* processor, the communication cost is zero.

Apart from some restrained cases the task graph scheduling problem is NP-hard [6]. Thus, most researchers use heuristic approaches to solve the problem

for reasonable sizes of the task graph. Three categories can be distinguished: list–based, clustering–based and duplication–based heuristics. List–based heuristics assign priority levels to the tasks and map the highest priority task to the best fitting processing element [8]. Clustering–based heuristics embrace heavily communicating tasks and assign them on the same processing element, in order to reduce the overall communication overhead[1]. Duplication-based heuristics also decrease the amount of communication while simultaneously the amount of (redundant) computation will be increased. It has been combined with both list-based [2] and cluster-based approaches [7]. In order to evaluate the quality of all those heuristics in a unified manner it would be desirable to compare the resulting schedules lengths of those heuristics to the *optimal* values.

In this paper, we describe an informed search method for computing optimal schedules for the task graph scheduling problem. It is based on the well known A*–Algorithm [9] and extended to the a space–saving variant, namely the IDA*–Algorithm. We used this algorithm to compute optimal schedules for task graphs with up to 24 tasks. In dependance of the task graphs' structure, the algorithm could be used for even larger graphs.

The paper is organized as follows. In the next section we introduce the problem representation by means of a decision tree. Then, we describe the sequential and parallel versions of the algorithm. Finally, we present and discuss our results and conclude the paper.

2 Problem representation by a decision tree

The task graph problem can be represented by a tree of states. At the root of that decision tree no tasks have been scheduled at all. Every intermediate node corresponds to a partial schedule. This means that a specific number of tasks is already assigned to the processing elements and that the starting times for those tasks are already determined as well.

Due to the dependencies of the remaining tasks, usually only a subset of these tasks can be scheduled in the next step. A specific node of the decision tree is expanded by creating new nodes for every ready task in combination with the mapping to the available target processing elements (TPE). If each of these tasks can be assigned to one of p target processing elements the total number of expanded nodes will be p times the number of ready tasks.

When all the internal nodes have been expanded this way we will get the leaves of the decision tree that represent valid schedules for the corresponding task graph. Obviously, the path length from the root to a leaf (the depth of the tree) is equal to the number of tasks. Each leaf is characterized by a specific schedule and a corresponding schedule length. The leaves with the minimal schedule length represent the optimal solution to the task graph problem.

As one can imagine, the decision tree comprises a huge number of states. Suppose n denotes the number of tasks. In the worst case, we have to expand $p{\cdot}n$ nodes from the root node, $p{\cdot}(n-1)$ nodes from each node of depth 1, $p{\cdot}(n-2)$ from each node of depth 2 and so on. Finally, after n expansions we get p leaves

from each node of depth n. In total there will be $1 + p^n \cdot n!$ states in the decision tree. Obviously, in the worst case there are no dependencies between the task. This means that the number of states will decrease as soon as we consider *true* task graphs with more constraints due to data dependencies. Nevertheless, the term p^n will persist and thus the task graph problem will be intractable.

The computation of the optimal schedules requires the construction of the decision tree's leaves. This can be accomplished by means of search algorithms. In order to accelerate the search for an optimal solution it will be useful to use *informed* instead of blind search algorithms. While a blind search generates the complete decision tree, the informed search uses additional information about the partial schedules in order to reduce the number of states that must be considered. One of the most popular informed search algorithms is the *A*–Algorithm* [9]. The A*–Algorithm requires that the whole decision tree is stored in memory. Thus, the space complexity is as worse as the time complexity and therefore we have to reduce the space complexity further. In this paper, we propose to use a modification of the A*–Algorithm that expands the nodes of the decision tree in a depth–first manner which is called *iterative deepening A*–Algorithm (IDA*)* [9]. By means of IDA* the space complexity can be reduced from $O(p^n)$ to $O(p \cdot n)$. While the A*–algorithm has already been applied to task graph problems [3, 5] — to our knowledge — the IDA*–algorithm hasn't yet been used to solve the task graph problem so far. In the next section we will give a short introduction to both of these algorithms.

3 A*– and IDA*–algorithm

Compared to an exhaustive (blind) search, the A*–algorithm reduces the number of the decision tree's nodes that must be expanded. Thus, the optimal schedules can be found in shorter time. In the context of task scheduling, the basic idea of the A*–algorithm is to use a function f that estimates the remaining schedule length of all schedules that originate from a specific partial (incomplete) schedule.

The function f is composed of two subfunctions g and h. For a specific state s of the decision tree, $g(s)$ represents the partial schedule length so far. $h(s)$ estimates the remaining time to a complete schedule that originates from the partial schedule specified by state s. h is called a *heuristic* function. If it *over*estimates the remaining time it can be shown that the A*–algorithm will find all optimal schedules for the task graph problem under investigation. In the case of task graph scheduling we can use the so called *static bottom–level (sbl)* to define such a heuristic function. The $sbl(i)$ of a task i can be computed as the maximum of summing up all working loads between that task and a terminal task while the communication costs between the tasks are ignored. We define the heuristic function $h(s)$ as follows:

$$h(s) := \max_{s'} sbl(s')$$

Here, s' denotes a successor task of the partial schedule s. Obviously, $h(s)$ overestimates the remaining costs and thus it can serve as a *admissable* function, e.g. a heuristic function that guarantees to find the complete and optimal solutions by means of the A*–algorithm.

Additionally, in order to ensure the monotonicity of the function $f(s)$ we define this function as

$$f(s) = \max_{s'}(f(s'), g(s) + h(s))$$

Where s' denotes any of the ancestor states of the partial schedule s.

The A*–algorithm works as follows: First, for each combination of a ready task and the available processing elements, a node in the decision tree is created. Then, the corresponding f-values are computed and we expand the node with the smallest f-value. This procedure will be repeated with all the expanded nodes until we get complete schedules (leaves of the decision tree). Notice, that we will get p schedules for each complete run through the decision tree because there are p processing elements.

In order to further reduce the computational effort of the A*–algorithm we propose to use an estimate f^* for the optimal schedule length. This estimate can be easily computed by any heuristic scheduling algorithm or by the first depth–search to a leaf of the decision tree (as described above). If $f(s) > f^*$ for any expanded state s it can be deleted because all ancestors of the corresponding partial schedule will have a schedule length that is worse than optimal. This avoids a lot of computation because the total number of evaluations is reduced. As already shown, the space complexity of the A*–algorithm is $O(p^n)$. Due to the resulting memory requirements, the application of the A*–algorithm is restricted to task graph scheduling problems with a quite small number of tasks.

This restriction can be softened by using an *iterative deepening* search. The corresponding IDA*–algorithm doesn't simultaneously expand all the partial schedules but it executes separate depth–searches. The space complexity decreases to $O(p \cdot n)$. Additionally, we also apply the estimation function f^* to eliminate redundant partial schedules as soon as possible. By means of this improved IDA*–algorithm, we were able to process task graph scheduling problems up to 24 tasks mapped to 10 TPEs. If we suppose only one byte per state a conventional A*–algorithm would require an inconceivable amount of 10^{24} byte in the worst case. In contrast, the memory requirements of the IDA*–algorithm can be easily fulfilled.

4 Parallelizing the IDA*-Algorithm

Our parallel IDA*-Algorithm was developed and tested on a PC-Cluster with 32 Computers (Athlon 800 MHz-Processors) using Linux and PVM. The program is started by a call of the master-module, which starts the required number[1] of slave-modules automatically. The parallelisation of the IDA*-Algorithm requires

[1] This number is given by the user.

a further subdivision of the search-space into disjunct subspaces, which can be assigned to the slave processes.

As already described in Section 2, every inner node of the decision-tree represents a partial schedule and every leaf node corresponds to a complete schedule. The algorithm's decision-tree guarantees that the successors of a node will represent different partial schedules – a later reunification of the subtrees, rooting in these sons, is impossible. Therefore two subtrees of the decision-tree always represent disjunct subspaces of the search-space, if none of their roots is an ancestor of the other one. Another result of these thoughts is that every part of the search-space can unambiguously be identified by it's root-node.

In order to achieve a balanced assignment of the computation to the available processing units, the algorithm generates a workpool, containing a certain number of subtree-roots. This workpool is managed by a master-process, which controls the distribution of the tasks to the slave-processes.

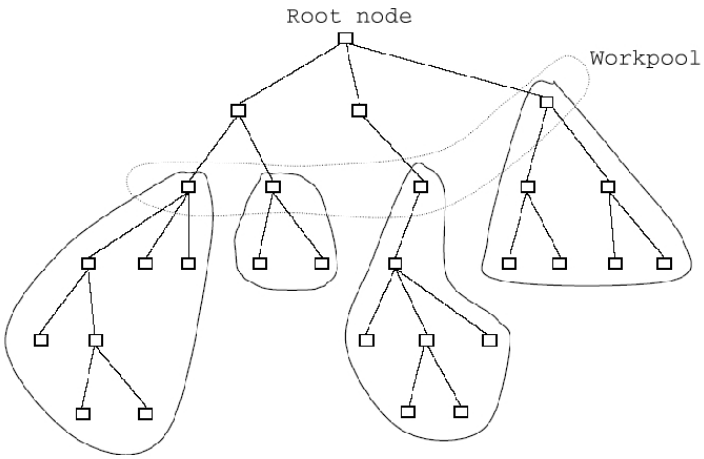

Fig. 1. Example of a small workpool

The workpool is created by means of a breadth-first-search which is halted, when a user defined number of elements is collected in the workpool.

Creation and management of the workpool are just two of the master's tasks which are embedded in several other duties, like management of the slaves and file I/O-Control. The master firstly reads the given parameters and the task graph. Then it generates the workpool and starts the slave processes. Next, it forwards one of the workpool's tasks to every slave process. As soon as one slave finishes it's task, the master receives and evaluates the slave's results. If the workpool is not empty, the master forwards another task to the idle slave and waits for further slaves to finish their work. Else, the master terminates the idle slave and tests, if there are any busy slaves left. If so, it waits for further slaves

to finish their work, else it prepares some bookkeeping information, creates the output-file and terminates.

The slave processes can be described more easily: When started, every slave process waits for an initial subspace to search. The search-process is realized by the means of the sequential IDA*-algorithm. When a slave finishes it's work, it sends the computed results to the master and waits for the next piece of work.

5 Results

5.1 Simplifying the search with a first estimate

Depending on the task graph's structure, the search space can become very huge even for small graphs. The significant strength of an informed search algorithm like the popular A* or the proposed IDA*-algorithm is their ability to use gained knowledge to reduce the search space's size as much as possible. The already described h-function is such an information, which can reduce the search space to a large extend. For the task graph given in figure 2, the h-function excludes 5 times as many partial schedules which have to be observed by the algorithm.

Algorithms like A* can be improved by using a first estimate of the real result for reducing the search space's size. This value has to be an overestimation of the optimal schedule length, because otherwise the algorithm would fail to provide a correct output. In contrast to the A*-algorithm, the IDA*-algorithm can not profit by using such an estimation.

We will illustrate this observation by regarding the task graph used as example in [10]. The schedules are computed for a target system's size of nine processors. For this system size, the optimal schedule length is 16 time units.

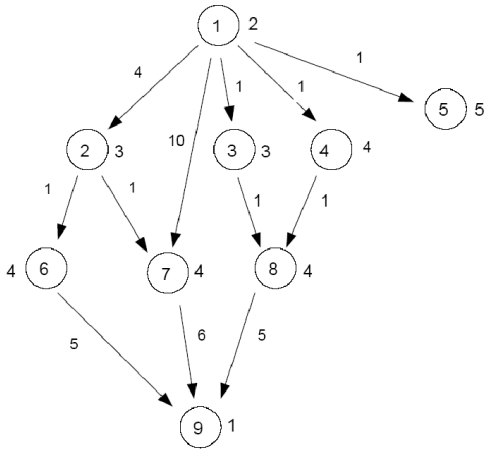

Fig. 2. The investigated task graph [10]

As it can be seen in the left part of figure 3, the number of observed partial schedules is almost independent of the estimation value. Hence, the runtime of the algorithm stays constant, see the right part of figure 3. The A*-algorithm shows a completely different behavior: if the estimation is near the optimal solution, the algorithm is faster than IDA*, but with a decreasing quality of the estimate, the size of the search space and therfore the runtime increases at a high rate.

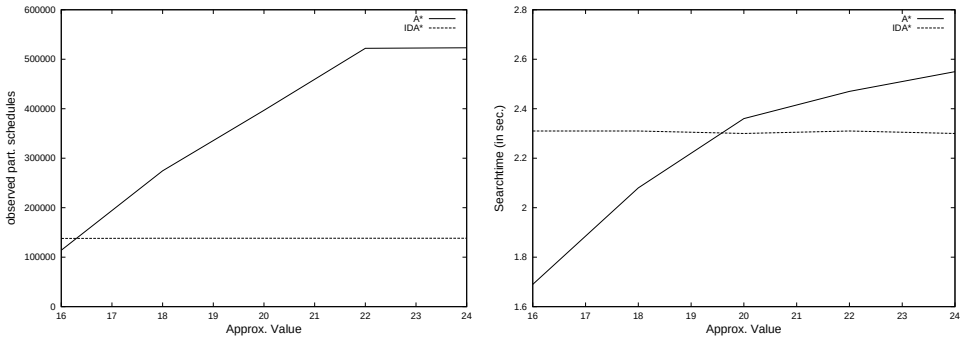

Fig. 3. The influence of a good first estimate

This observation can be explained as follows. Considering the approximation value of 16 time units, the IDA*-algorithm has to search more partial schedules than A*, because it additionally has to process all partial schedules with a partial schedule length of 16 to ensure, that 16 really is the optimum. IDA* is immune to changes of the initial estimation because it adapts every newly found best schedule length at once.

Obviously, the A*-algorithm's performance can be highly improved by using a heuristic for computing an estimate of the expected result. If the estimation is near the optimal solution, A* performs better than the here proposed IDA*-algorithm. Else, IDA*, which is not affected by an initial guess at all, performs much better than A*. Since the IDA*-algorithm shows a good overall performance, it is a good choice especially for those scheduling problems where no estimate of the optimal schedule length exists.

5.2 Effects of the target's system size

The target architecture system's size influences two significant aspects of our algorithm: the time required to compute an optimal schedule and the optimal schedule length itself. Although both dependency-relations are known in general, every algorithm behaves different concerning the first one. To demonstrate our algorithm's behaviour, we use the task graph shown in figure 2 again.

Table 1 shows the behavior of our algorithm when the number of TPEs changes. The optimal schedule length can be achieved by using 3 processing

elements – an enlargement of the target system would not improve the performance. The maximum number of used TPEs is 6 – forcing the algorithm to use more processors would lead to a slowdown. This value is determined by the task graph's structure: the arising communication cost will outweight the advantage of any additional parallelization.

Table 1: Effect of changing the target system's size

TPEs	opt. SL	TPEs used	part. Complete schedules	Part. Schedules	Runtime
1	30	1	640	1721	0.02
2	17	2	675	11879	0.15
3	16	3	3021	52917	0.71
4	16	4	11103	100757	1.49
5	16	5	15843	132150	2.11
6	16	6	16461	137915	2.33
7	16	6	16462	138273	2.31
8	16	6	16463	138275	2.28
9	16	6	16464	138275	2.30

As one can see in the last three columns of table 1, the increasing number of TPEs enlarges the algorithm's decision tree and therefore the efforts to find an optimal schedule. Additionally it is obvious, that the maximum system size is limited to 6 processing elements. The estimating function h prevents the usage of further processing elements and hence shrinks the search space.

5.3 Application

Considering, that task graph scheduling is a NP-hard problem, the question arises, why someone might be interested in a scheduling algorithm, which provides optimal solutions, but only for small task graphs.

We used the proposed algorithm to compute a database of 3600 optimal schedules for randomly generated task graphs. The task graph's size is equally distributed between 7 and 24 tasks. This test base is structured by some of the task graphs' properties, like the meshing degree, or the nodes' and edges' weights.

The first purpose for creating this test bench was the creation of an objective baseline for the evaluation of scheduling heuristics. Up to now, scientists evaluate their heuristics by simply comparing their algorithm's results with those of other, already know heuristics. This method does neither guarantee an unbiased selection of the used test cases, nor does it provide any information about the absolute heuristic's quality. With an comprehensive test bench, one will be able to create more reliable and expressive evaluations. Currently, a test bench with 36000 test cases with up to 24 tasks is generated which will soon be published for this purpose. As presented in [4], these task graphs are large enough to provide a sophisticated analysis of scheduling heuristics.

The second purpose for creating this test bench is to test the reliability of other algorithms which promise to calculate optimal schedules. Although it is impossible to guarantee an implementation's correctness, this test bench can be used to check an algorithm's quality. The test bench itself was validated by a branch-and-bound-algorithm [4], which computed the same optimal schedule lengths in all cases. To improve the reliability of this validation, the second algorithm was implemented by a different programmer. Today, we use this test bench to verify, that improvements of these algorithms do not effect the optimality of the computed results. Interestingly, the smaller task graphs of our test bench are more sensitive to incomplete search algorithms than the larger ones. This can be explained by the huge number of optimal solutions, most larger task graphs possess. It is therefore less likely that their optimal solution is overlooked.

6 Conclusion

In this paper, we presented a parallel implementation of the IDA*-algorithm for computing optimal task graph schedules. This informed search algorithm is faster than any exhaustive search and less memory consuming than the well known A*-algorithm. In combination with the parallelization, this allows the computation of more complex task graph schedule's in reasonable time.

In contrast to A*-, the IDA*-algorithm has been almost insensitive to any given initial estimate of the schedule length. The size of the considered target architecture affects the size of the resulting search space. Thus, the runtime of the IDA*-algorithm depends highly on the chosen target architecture. Although the algorithm's runtime prohibits its online application it nevertheless can be used to create a test base for the evaluation of scheduling heuristics and the verification of future algorithms which promise near optimal results.

The described algorithm is currently in use to compute the optimal schedules for a benchmark suite that comprises 36,000 task graph problems with up to 24 tasks. In the near future, this benchmark suite will enable researchers to evaluate the performance of their heuristics with the actually best solutions.

7 Acknowledgment

The authors would like to thank Mr. Johann Zeiser who contributed some of the presented results from his diploma thesis.

References

1. Aguilar, J., Gelenbe E.: Task Assignment and Transaction Clustering Heuristics for Distributed Systems, Information Sciences, Vol. 97, No. 1& 2, pp. 199–219, 1997
2. Bansal S., Kumar P., Singh K.: An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems, IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 6, June 2003
3. Dogan A., Özgüner F.: Optimal and Suboptimal reliable scheduling of precedence-constrained tasks in heterogeneous distributed computing, International Workshop on Parallel Processing, p. 429, Toronto, August 21-24, 2000
4. Hönig U., Schiffmann W.: A Parallel Branch–and–Bound Algorithm for Computing Optimal Task Graph Schedules, Second International Workshop on Grid and Cooperative Computing, pp. 747–755, Shanghai, Dec. 7-12, 2003
5. Kafil M., Ahmad I.: Optimal Task assignment in heterogeneous distributed computing systems, IEEE Concurrency: Parallel, Distributed and Mobile Computing, pp. 42-51, July 1998
6. Kwok, Y.-K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys, Vol. 31, No. 4, 1999, pp. 406–471
7. Park C.-I., Choe T.Y.: An optimal scheduling algorithm based on task duplication, IEEE Transactions on Computerss, Vol. 51, No. 4, April 2002
8. Radulescu A., van Gemund A. J.C.: Low-Cost Task Scheduling for Distributed-Memory Machines, IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 6, June 2002
9. Russel S., Norvig P.: Artificial Intelligence, A Modern Approach, Prentice Hall, 1995
10. Kwok, Y.-K.: High-Performance Algorithms for Compile-Time Scheduling of Parallel Processors PhD-Thesis, University of Hong Kong, 1997