A Comparison of Scheduling Algorithms for Multiprocessortasks with Precedence Constraints

Jörg Dümmler, Raphael Kunis and Gudula Rünger
Chemnitz University of Technology
Department of Computer Science
09107 Chemnitz, Germany
E-mail: {djo,krap,ruenger}@cs.tu-chemnitz.de

Keywords— **Multiprocessortask Programming, Scheduling, Distributed Memory, Scalable Computing**

Abstract— **Many parallel applications from scientic computing show a modular structure and are therefore suitable for the multiprocessortask programming model with precedence constraints. This programming model has been shown to yield better results than a pure dataparallel or a pure taskparallel execution on distributed memory platforms in many cases. The efficient execution of multiprocessortask programs requires an appropriate schedule, which takes the structure of the application and the performance characteristics of the target platform into account. Many heuristics and approximation algorithms have been proposed to fulfil this scheduling task. In this paper we consider popular scheduling algorithms that have been implemented in a scheduling toolkit. Specifically, we introduce Allocation-and-Scheduling-based algorithms and compare their runtime for large task graphs consisting of up to 1000 nodes and target systems with up to 256 processors. Furthermore we consider the quality of the produced schedules and derive a guideline describing which scheduling algorithm is most suitable in which situation.**

I. Introduction

A current challenge in the development of parallel applications for distributed memory platforms is the achievement of a good scalability even for a high number of processors. Often the scalability is impacted by the use of collective communication operations like broadcast operations, whose runtime exhibits a logarithmic or even linear dependence on the number of participating processors. Especially the advent of large homogeneous cluster systems and hierarchical cluster-of-clusters, which consist of multiple homogeneous clusters, necessitates a programming model that can help to reduce the communication overhead.

A possible approach is the model of multiprocessortask (short M-Task) programming with dependencies [7]. In this programming model a parallel application consists of a set of M-Tasks, which can be executed on an arbitrary subset of the available processors. Additionally there may be dependencies between M-Tasks meaning that these M-Tasks have to be executed one after another. These dependencies usually arise from communication phases that are necessary between the execution of M-Tasks to exchange data. For independent M-Tasks a consecutive or a concurrent execution on disjoint subsets of the available processors is possible. A schedule assigns each M-Task at least one processor and fixes the execution order of the M-Tasks. For a given M-Task program many different schedules may be possible. Which schedule achieves the best results, i.e. leads to a minimum parallel runtime of the application, depends on the application itself and on the target platform. Therefore for target platforms with different computation and communication behavior different schedules may lead to a minimum runtime. Determining the optimal schedule is an NP-hard problem, but many scheduling heuristics and approximation algorithms have been proposed to get a near optimal solution to this problem.

Developing an M-Task application is more complex and error-prone compared to the development of pure SPMD applications. This mainly results from two different types of communication (between M-Tasks vs. within an M-Task) and from the additional code required to manage the partitioning of the set of processors into subsets, on which the M-Tasks are executed. Furthermore, changes in the schedule of an M-Task application usually require a complex restructuring of the whole program resulting from a different processor group layout and a different communication pattern. A variety of languages, tools, libraries and frameworks to assist in the development of M-Task applications has been proposed. An overview is given in [1]. Most of these approaches still require the developer to manually specify the schedule for an application. As different target platforms may require different schedules this leads to a poor portability.

Many of the proposed scheduling algorithms for M-Task applications with precedence constraints have similarities on how to approach the scheduling problem. We distinguish three main categories, which we call *Allocation-and-Scheduling-based* algorithms, *Layer-based* algorithms, and *Configuration-based* algorithms. *Allocation-and-Scheduling-based* algorithms consist of an allocation step, which fixes the number of executing processors for each M-Task, and a scheduling step, which determines the execution order of the tasks and the exact processor groups. *Layer-based* algorithms shrink and decompose the directed acyclic graph representing an M-Task application into a set of layers of independent M-Tasks. The scheduling is performed for each layer in isolation and the resulting layer schedules are joined into a global schedule for the whole application. *Configuration-based* algorithms are single step methods that construct schedules based on a predefined set of possible configurations for each M-Task.

In this paper we examine *Allocation-and-Scheduling-based* algorithms. We present a detailed comparison of the runtime and the quality of the produced schedules for scheduling algorithms from this class. Finally, we derive a guideline, which M-Task scheduling algorithm is most suitable in which situation. The performance of M-Task scheduling algorithms was compared in [4], [5], however only small applications were considered and a different set of algorithms was used. To enable an automatic scheduling of M-Task applications based on cost expressions for the M-Tasks and for the communication between the tasks we develop a scheduling toolkit[2]. This toolkit includes scheduling algorithms from different categories including the presented *Allocation-and-Scheduling-based* algorithms. The use of a toolkit permits the examination of different scheduling algorithms using an identical environment by utilising similar data structures for representing M-Task applications and schedules.

This paper is structured as follows. Section II explains the multiprocessortask programming model with dependencies. Section III gives an overview of the considered scheduling algorithms. The obtained benchmark results are discussed in Section IV and Section V concludes the paper.

II. Programming Model

Many programming models are based on M-Tasks. In the M-Task programming model with precedence constraints a parallel application can be represented by an annotated direct acyclic graph (M-Task dag) $G = (V, E)$. Figure 1 shows an example of an M-Task dag.

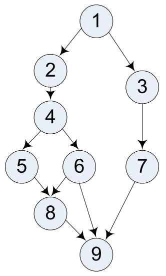

Fig. 1. Example of a small M-Task dag.

A node $v \in V$ corresponds to the execution of an M-Task, which is a parallel program part that can be executed on any nonempty subset $g_v \subseteq \{1, \ldots, P\}$ of the available processors of a P processor target platform. The size of a processor group $|g_v|$ is also denoted as the allocation a_v of a task v. The allocation $A : V \rightarrow [1, \ldots, P]^{|V|}$ of the M-Task dag unites the single allocations a_v $\forall v \in V$.

A directed edge $e = (v_1, v_2) \in E$ represents precedence constraints between the M-Tasks v_1 and v_2, i.e. v_1 produces output data required for v_2 and therefore v_1 and v_2 have to be executed one after another. Edges may lead to a data redistribution if the processor group changes, i.e. $g_{v_1} \neq g_{v_2}$ or if v_1 and v_2 require different data distributions. M-Tasks that are not connected by a path in the M-Task dag can be executed concurrently on disjoint subsets of the available processors. Each node $v \in V$ is assigned a computation cost $T_v : [1, \ldots, P] \rightarrow \mathbb{R}^+$ and each edge $e = (v_1, v_2) \in E$ is assigned a communication cost $T_{comm}(v_1, v_2)$.

The costs for a path in the M-Task dag under a given allocation are defined as the sum of the computing costs of all nodes and the data redistribution costs of all edges belonging to the path. The longest path in the M-Task dag is called the *critical path* and has a length of $T_{CP}(A)$. The set $CP(A)$ contains all nodes on the critical path. The *top level* $TL_v(A)$ of an M-Task $v \in V$ is the length of the longest path from any task without predecessors to task v excluding v. The length of the longest path from a node $v \in V$ to any node without successors including v is called the bottom level $BL_v(A)$. The work W_v of an M-Task $v \in V$ is the product of the computing time and the allocation, i.e. $W_v(a_v) = T_v(a_v) * a_v$. The average computing area T_A is the arithmetical mean of the works of all M-Tasks, i.e. $T_A(A) = \frac{1}{P} \sum_{v \in V} W_v(a_v)$.

An M-Task may either be a basic M-Task that is implemented using an SPMD programming style or a complex M-Task that is built up from other M-Tasks and can be represented by an M-Task dag. Hence a hierarchical structure as it is described in the TwoL(Two Level)-approach[8] arises.

The execution of an M-Task application is based on a schedule S, which assigns each M-Task $v \in V$ a processor group g_v and a starting time T_{S_v}, i.e. $S(v) = (g_v, T_{S_v})$. A feasible schedule has to assure that all required input data are available before starting an M-Task, meaning that all predecessor tasks have finished their execution and all necessary data redistributions have been carried out, i.e.

$$T_{S_n} + T_n(|g_n|) + T_{comm}(n, m) \leq T_{S_m}$$
$$\forall n, m \in V \text{ and } (n, m) \in E.$$

Furthermore M-Tasks whose execution time interval overlaps have to run on disjoint processor groups, i.e. if

$$[T_{S_n}, T_{S_n} + T_n(|g_n|)] \;\cap\; [T_{S_m}, T_{S_m} + T_m(|g_m|)] \neq \emptyset$$
$$\text{then } g_n \cap g_m = \emptyset \;\; \forall\, n, m \in V.$$

The makespan $C_{max}(S)$ of a schedule S is defined as the point in time at which all M-Tasks of the application have finished their execution, i.e.

$$C_{max}(S) = \max_{v \in V}(T_{S_v} + T_v(|g_v|)).$$

In this paper we only consider non-hierarchical M-Task applications, i.e. only basic M-Tasks are available, and we do not take data redistribution costs into account. This is feasible as these costs are usually a magnitude lower compared to the computational costs of the M-Tasks and it is often possible to hide at least parts of these costs by overlapping of computation and communication. We only consider M-Tasks graphs that belong to the class of series parallel graphs (sp-graphs), because these graphs reflect the regular structure of most scientific applications. Series-parallel-graphs are a subset of directed acyclic graphs that are built by the following recursive definition [9]. A single node is an sp-graph. Two sp-graphs $SP_1 = (V_1, E_1), SP_2 = (V_2, E_2)$, can be combined to a new sp-graph by a series composition or a parallel composition. A sink node of a sp-graph is a node without successors and a source node is a node without predecessors. A series composition connects every node from the set of sinks $T_1 \subseteq V_1$ of SP_1 with nodes from the set of sources $S_2 \subseteq V_2$ of SP_2 by a new edge: $SP_{new} = (V_1 \cup V_2, E_1 \cup E_2 \cup T_1 \times S_1)$. A parallel composition merges the set of nodes and the set of edges of SP_1 and SP_2 to a new series-parallel graph: $SP_{new} = (V_1 \cup V_2, E_1 \cup E_2)$.

III. Scheduling Algorithms

A popular approach to the M-Task scheduling problem with precedence constraints is a two-step approach introduced in [6] consisting of an allocation step and a scheduling step, which we therefore call *Allocation-and-Scheduling-based* algorithms. The allocation step determines the allocation a_v for each node $v \in V$ of the M-Task dag. The exact layout of the processor group g_v and the starting time index T_{S_v} is determined in the scheduling step. Most of these algorithms only differ in the allocation step and use a *modified List-Scheduling* algorithm in the scheduling step.

The modified List-Scheduling algorithm is based on a priority queue with different priority functions (earliest start time, bottom level, top level, smallest task or largest task). The List-Scheduling algorithm works as follows. Initialize the priority queue q by adding the source nodes of the M-Task dag. While q is not empty remove and schedule the head node v of the queue. The scheduling of v with an actual start time of T_{S_v} is done by finding a suitable set g_v of processors for a_v with the earliest processor ready time T_{g_v}. Afterwards the schedule time index T_{S_v} is re-evaluated by the following equation:

$$T_{S_v} = max(T_{g_v}, T_{S_v})$$

The finish time $T_{F_v} = T_{S_v} + T_v(a_v)$ of v is computed and the earliest start time of the successors of v and the processor ready times of the processors in g_v are set to this time. If the scheduling of v leads to the fulfilment of all precedence constraints of a successors of v this successor is added to the priority queue q. The worst case complexity of the modified List-Scheduling algorithm is $\mathcal{O}(E + Vlog(V) + VP)$ resulting from $\mathcal{O}(V + E)$ for the computation of the task priorities, $\mathcal{O}(VlogV)$ for removing V tasks from the queue and maintaining the queue ordering and $\mathcal{O}(VP)$ to schedule V tasks on P processors.

In the following we present the main ideas of scheduling algorithms belonging to the class of *Allocation-and-Scheduling-based* algorithms.

a) **Dataparallel:** The *Dataparallel* scheduler allocates all available processors to each M-Task in the allocation phase resulting in an SPMD processing style. A topological sort of the task graph can be used to obtain an order of execution of the tasks in the scheduling step. As the nodes of the input task graphs in our scheduling toolkit are already stored in topological order, the *Dataparallel* scheduler runs in $\mathcal{O}(V)$.

b) **Taskparallel:** The *Taskparallel* scheduler produces a schedule as it is known from uniprocessortask scheduling. The allocation phase assigns each M-Task a single processor and a List-Scheduling algorithm with the bottom levels as a priority function is used for the scheduling phase. The worst case time complexity of $\mathcal{O}(E + VlogV + VP)$ for the *Taskparallel* scheduler is equal to the List-Scheduling algorithm.

c) **TSAS:** The *Two Step Allocation and Scheduling(TSAS)*[6] scheduler transfers the problem to find a discrete allocation for each M-Task to an optimization problem in the continuous space. The objective of the optimization is to find an allocation $A_c : V \rightarrow \mathbb{R}^{|\mathbb{V}|}$ that minimizes $\max\{T_{CP}(A_c), T_A(A_c)\}$. The intention is to find an allocation that is a good trade-off between critical path length and average area, which are both lower bounds on the makespan of any feasible schedule. If the cost functions are posynomials a convex optimization problem results, which has a unique global minimum that can be determined by an iterative algorithm in polynomial time. A posynomial function f of a positive vector variable $x \in \mathbb{R}^m$ has the form

$$f(x) = \sum_{i=1}^{N} c_i \prod_{j=1}^{m} x_j^{a_{ij}}$$

with positive coefficients $c_i \in \mathbb{R}^+$ and exponents $a_{ij} \in \mathbb{R}$. For our tests we use cost expressions based on Amdahl's law, which are posynomial functions. The result of the allocation phase is obtained by mapping the continuous solution of the optimization problem to discrete space. The scheduling phase uses a List-Scheduling algorithm with the earliest possible start time of a task as a priority function.

d) **CPA:** The *Critical Path and Area-based scheduling (CPA)*[5] scheduler was designed as a low-cost scheduler and a computationally cheap heuristic is employed for the allocation phase. The idea of the allocation phase is to find an allocation A that minimizes $\max\{T_{CP}(A), T_A(A)\}$ and is therefore similar to *TSAS*. The starting point is an allocation of one processor per task, i.e. $a_v = 1 \ \forall v \in V$. Each iteration of the main iteration loop chooses a critical path task $v \in CP(A)$ and increases its allocation, i.e. $a_v = a_v + 1$. As a consequence, the critical path may change and other nodes are considered in subsequent iterations. The main loop terminates when the average area exceeds the length of the critical path for the current allocation, i.e. $T_A(A) \geq T_{CP}(A)$. The scheduling phase uses a List-Scheduling heuristic with the bottom levels of the tasks as a priority function. The worst case complexity of *CPA* is $\mathcal{O}(V(V + E)P)$, which arises from $\mathcal{O}(VP)$ iterations of the main iteration loop each requiring $\mathcal{O}(V + E)$ time to compute the critical path and a single execution of the List-Scheduling algorithm.

e) **CPR:** The *Critical Path Reduction (CPR)*[4] scheduler follows a similar approach as the *CPA*-scheduler but uses a more complex heuristic in the allocation phase to reduce the length of the critical path in the M-Task dag. *CPR* starts with an allocation of one processor per task, i.e. $a_v = 1 \forall v \in V$. The main iteration loop first computes a priority *prio* for all tasks based on the sum of the top and bottom level, i.e. $prio_v = TL_v + BL_v \forall v \in V$ and inserts all tasks in a priority queue. Afterwards the head task v of the priority queue is removed from the queue, the allocation is increased by 1, i.e. $a_v = a_v + 1$, and a List-Scheduler with a bottom level priority function is run with the current allocation. If the constructed schedule has a lower makespan than any previously obtained schedule, the current allocation is commited and the main loop is started over. Otherwise the changes are rejected, i.e. $a_v = a_v - 1$, and the next task in the priority queue is considered. The main loop terminates, when the priority queue runs empty, i.e. there is no task for which increasing its allocation results in a better schedule. The time complexity of *CPR* is $\mathcal{O}(EV^2P + V^3Plog(V) + V^3P^2)$ and results from $\mathcal{O}(VP)$ executions of the main iteration loop in the worst case, which occurs when P processors are allocated to each of the V tasks. Each loop iteration may have to call the List-Scheduling algorithm for all V tasks in the worst case.

f) **MSAA:** The *Modified sp-graph approximation algorithm (MSAA)*[3] scheduler uses an approximation algorithm based on integer values for the execution time of the tasks in the allocation phase and an earliest start time List-Scheduler in the scheduling phase. The approximation algorithm tries to decide within pseudo-polynomial time whether an allocation with costs $c(A) \leq X$ exists for a given positive integer bound X. X represents the critical path length in the M-Task dag for a pure taskparallel allocation

$$c(A) = max\{T_{CP}(A), T_A(A)\} \leq C_{max}(S).$$

This algorithm operates not at the sp-graph itself but on the decomposition tree of the sp-graph. The sp-graph decomposition tree $G_D = (V_D, E_D)$ corresponds to a rooted, ordered, binary tree [9]. The internal nodes correspond to the composition of the sp-graph and are labeled s (series composition) or p (parallel composition). The leafs are the nodes in the sp-graph. After the decomposition of the sp-graph a matrix F of dimension $|V_D| \times X$ is built that contains the values $F[v_D, l], 1 \leq v_D \leq |V_D|, 0 \leq l \leq X$. Each $F[v_D, l]$ represents the smallest possible value for the work $W_{v_D}(A_{v_D})$ of a node v_d in the decomposition tree that holds the following property: An allocation A for the tasks in the sub decomposition tree under v_d exists with

$$T_{CP}(A) \leq l, T_A(A) \leq W_{v_D}(A_{v_D}).$$

A dynamic programming approach is used to compute all $F[v_D, l]$ values starting in the leafs of the decomposition tree and moving upwards to the root. The last step of the algorithm is to find a value for l with $F[root_D, l]/P \leq X$. Because more than a single l can be found, we use all possible candidates of l in the list scheduling step to determine the best solution. The allocations for the l-values can be found by storing additional information in the computing step of all values in F. The complexity of the allocation algorithm is $\mathcal{O}(|V_D| * P * X^2)$. X^2 is the main factor in the complexity. We try to decrease the runtime of the algorithm by mapping the runtimes of the nodes to integers getting an X that is small but produces relative good schedules. The mapping has to be performed, because the algorithm needs integers to process the dynamic programming approach. We use the following formula to map the runtimes: $X = (1 + |V|/adapt) * 4, adapt = 250$. This mapping depends on the number of nodes $|V|$ in the sp-graph. It tries to find a good solution for getting different integer values for different original execution times of the tasks by keeping X small. The value for *adapt* is based on runtime tests on our target machine and it is possible to find a good solution for *adapt* by running some tests on other target machines.

IV. Results

A. Testing Environment

The benchmark tests presented in this Section are obtained by running the scheduling toolkit on an Intel Xeon 5140 (“Woodcrest“) system clocked at 2.33 GHz. The available main memory was 8 GB cached by an L2 cache with a size of 4 MB. To run the scheduling toolkit the 64 bit version of the *Java 2 SE Runtime Environment (JRE)* Version 5.0 Update 9 was used.

For the benchmarks, we use *test sets* consisting of 100 different M-Task dags, which belong to the class of series-parallel-graphs (sp-graphs). The generation algorithm used to construct these graphs starts with a number of sp-graphs consisting of a single node and randomly combines these graphs by a serial or a parallel composition. Afterwards all nodes of the graph are annotated by a runtime estimation formula according to Amdahl's law ($T_{par} = (f + (1 - f)/p) * T_{seq}$), which describes the parallel runtime T_{par} on p processors for a problem with an inherent sequential fraction of computation $f (0 \leq f \leq 1)$ and a runtime on a single processor $T_{seq} (T_{seq} > 0)$.

B. Runtime Results

In this Subsection we consider the runtime of the implemented scheduling algorithms. All presented measurements are the arithmetical mean of the runtimes for each M-Task dag within a test set. The same test set was used when varying the number of processors, but changing the number of nodes requires a different set.

The *Dataparallel* and *Taskparallel* schedulers achieve the lowest runtimes of all scheduling algorithms as the execution does not involve a sophisticated scheduling process. The runtimes of the *Dataparallel* scheduler range from 0.5 ms for 50 nodes to 10.4 ms for 1000 nodes and are independent from the number of processors. These execution times can be considered as a general overhead factor for the management of the internal structures of the scheduling toolkit. Figure 2 shows the average runtimes of the *Taskparallel* scheduler. The measurements for the *Taskparallel* scheduler show an almost linear dependence on the number of nodes and a slow increase of the runtime with the number of processors. The runtime for 256 processors is about 20%

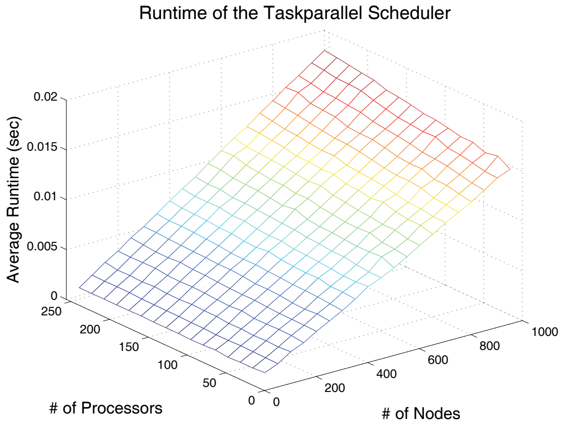

Fig. 2. Average runtime of the *Taskparallel* scheduler for varying number of nodes and processors.

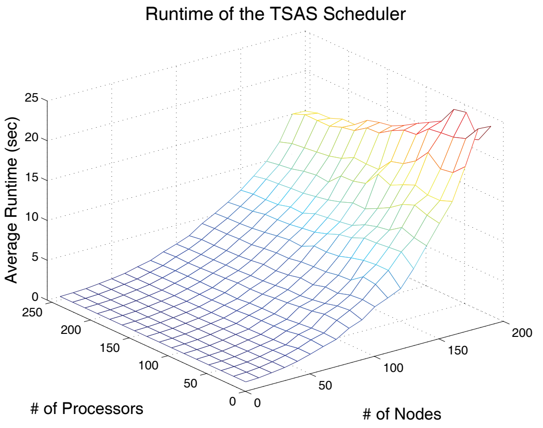

Fig. 3. Average runtime of *TSAS* for different number of nodes and processors.

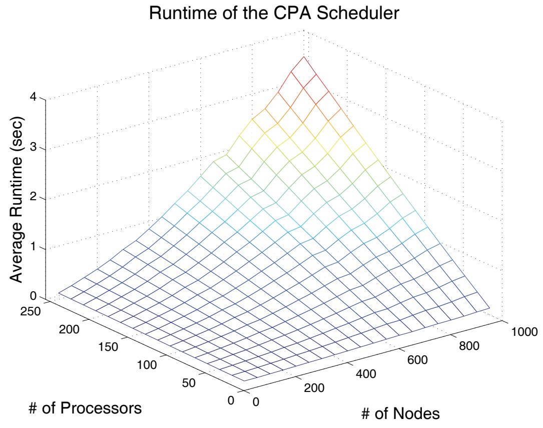

Fig. 4. Average runtime of *CPA* dependent on the number of nodes and processors.

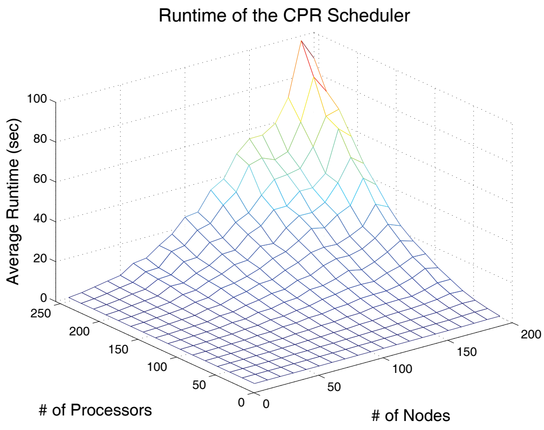

Fig. 5. Average runtime of *CPR* for varying number of nodes and processors

higher compared to the execution time for 16 processors. These runtimes are also a measure for the List-Scheduling part in *CPA*, *CPR*, *TSAS* and *MSAA*, as the same heuristic is employed in these algorithms.

The average runtimes of *TSAS* are shown in Figure 3. Because of the high runtime requirements for 100 test runs we only show results for up to 200 nodes for this scheduling algorithm. The main part of the work of *TSAS* is done in the allocation step, when solving the convex optimization problem. The runtime of this step is directly influenced by the number of required iterations. For all tested numbers of nodes the average runtime of *TSAS* decreased when increasing the number of processors. This is an interesting behavior that is unique within the tested scheduling algorithms and can be explained by a faster convergence of the convex programming approach. In our tests we experienced a medium deviation between the minimum and the maximum runtimes of *TSAS* for a given test set.

As the average execution times of *CPA* that are presented in Figure 4 show, *CPA* achieves a good performance even for a high number of nodes for target platforms with a low number of processors. The runtime increases linearly with the number of processors agreeing with the worst case complexity of *CPA*. Although the results show a constant increase of the runtime with the number of nodes and the number of processors, the runtimes within a given test set exhibit a large deviation. This mainly results from a different number of iterations in the main allocation loop, which is responsible for the biggest part of the runtime. The number of iterations can be between 1 (if the termination criterion is met with the initial allocation) and $\mathcal{O}(VP)$ in the worst case (when the resulting allocation assigns all processors to each node).

The average runtime results for *CPR* are shown in Figure 5. *CPR* exhibits the slowest average runtimes of all implemented scheduling algorithms and was therefore only tested for M-Task dags with up to 200 nodes. Compared to *CPA* the much higher runtime results from the execution of the List-Scheduling heuristic in each iteration, whereas *CPA* only requires a single List-Scheduling step. For target platforms with a low number of nodes *CPR* still achieves reasonably low runtimes. The runtime of *CPR* strongly depends on the input scheduling problem. The deviation between the minimum and maximum runtimes for a given test set is the highest of all scheduling algorithms. The reason for this behavior is similar to *CPA* a varying number of iterations in the main allocation loop, which can be between 1 and $\mathcal{O}(VP)$. This can also explain, why the test set with 190 nodes requires a higher average run-

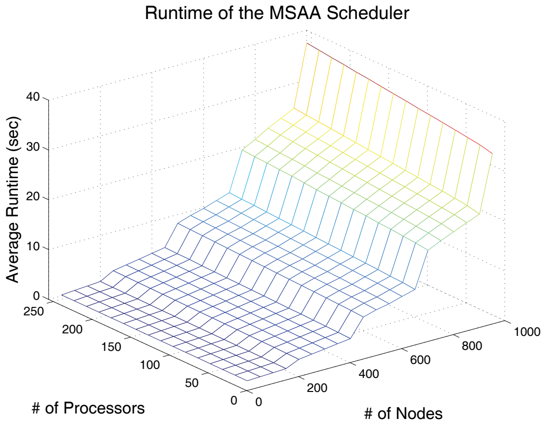

Fig. 6. Average runtime of *MSAA* for varying number of nodes and processors

TABLE I: Average Runtimes relative to the *Dataparallel* Scheduler.

Scheduler	200 nodes 16 proc.	200 nodes 256 procs.	1000 nodes 16 procs.	1000 nodes 256 procs.
Data	1	1	1	1
Task	1.36	1.67	1.41	1.71
TSAS	11484	7254	n/a	n/a
CPA	9.00	158	13.1	331
CPR	214	42215	n/a	n/a
MSAA	221	376	3126	3573

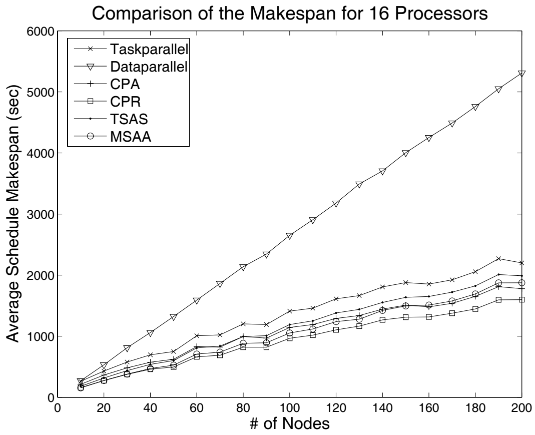

Fig. 7. Comparison of the average makespan of different scheduling algorithms for task graphs with 10 to 200 nodes and 16 available processors.

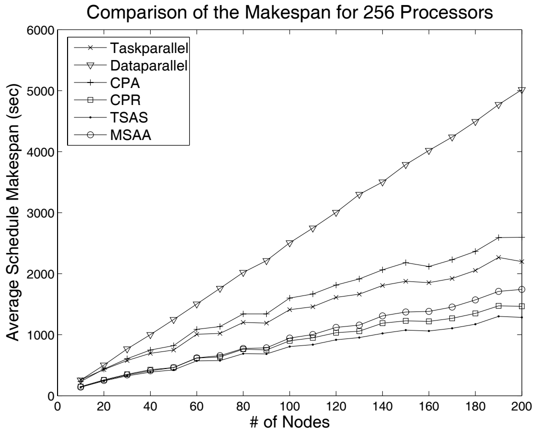

Fig. 8. Comparison of the average makespan of different scheduling algorithms for task graphs with 10 to 200 nodes and 256 available processors.

time compared to the test set with 200 nodes.

Figure 6 shows the runtime results for *MSAA* with critical path adaption. The results show that this algorithm achieves a good performance for small and high numbers of processors and nodes in comparison to *CPR* and *TSAS*. The dependency on the X-value can be seen in the jumps at 250, 500, 750 and 1000 nodes. The runtime is linear in the number of nodes and linear in the number of processors if X is fixed. This linear runtime results from the structure of the $|V_D| \times X$ matrix used for computing the F-values that is linear in $|V_D|$ and P and the main factor is X^2 in the allocation step. The runtimes within a given test set exhibit only small deviations. This results from the dependency of the runtime on X^2 which is much larger than $|V|$ and P.

Table I gives an overview of the relative runtimes of all tested scheduling algorithms compared to the *Dataparallel* scheduler averaged over 100 test runs. It comes to no surprise that the *Taskparallel* and *Dataparallel* Schedulers have a much lower runtime compared to the other algorithms as no sophisticated scheduling logic is involved. From the specialized scheduling algorithms *CPA* achieves the highest performance and especially for a low number of processors clearly outperforms all other algorithms. For a high number of processors the gap between *CPA* and *MSAA* becomes smaller and it can be assumed that *MSAA* beats *CPA* for processor numbers somewhat higher than 256. *TSAS* and *CPR* (for a high number of nodes) exhibit a considerably higher runtime than all other algorithms. *CPR* is faster than *TSAS* for a low number of processors, whereas *TSAS* beats *CPR* for larger target platforms.

If the runtime of the scheduling algorithm is an issue, *CPA* is a good choice, because it achieves the lowest runtimes of all specialized scheduling algorithms. For target systems with many processors *MSAA* might be a good choice since its runtime is also almost independent from the structure of the input problem.

C. *Quality of the Schedules*

In this Section we consider the quality of the produced schedules, i.e. the makespan based on the runtime estimation formulas. First we consider the average makespan for the test sets with up to 200 nodes. The results for all tested scheduling algorithms are shown in Figure 7 for 16 available processors and in Figure 8 for 256 processors respectively. Especially Figure 8 shows no crossing point between the curves of the scheduling algorithms, i.e. a scheduling algorithm that achieves a

TABLE II: Speedups of the produced schedules relative to the *Dataparallel* Scheduler.

Scheduler	16 procs.	64 procs.	128 procs.	256 procs.
Data	1	1	1	1
Task	1.89	1.81	1.79	1.79
TSAS	2.25	2.73	2.98	3.14
CPA	2.33	1.91	1.72	1.60
CPR	2.75	2.78	2.79	2.80
MSAA	2.51	2.68	2.65	2.62

TABLE III: Number of constructed schedules with the lowest makespan for 16 (left value) and 256 processors (right value).

Scheduler	50 nodes	100 nodes	150 nodes	200 nodes
Data	0/0	0/0	0/0	0/0
Task	0/0	0/0	0/0	0/0
TSAS	1/68	0/87	0/93	0/92
CPA	9/0	11/0	12/0	18/0
CPR	67/18	77/8	80/7	80/8
MSAA	23/14	12/5	8/0	2/0

TABLE IV: Recommended scheduling algorithms for different situations.

	low number of processors	high number of processors
low number of nodes	*CPR** *CPA***	*TSAS** *MSAA***
high number of nodes	*CPR** *CPA***	*TSAS** *MSAA***

* best quality ** good quality, reasonable runtime

better quality for a low number of nodes is also better for a higher number of nodes. The *Dataparallel* scheduler delivers the schedules with the highest makespans for all tested problem instances. The gap to the other scheduling algorithms increases with the number of nodes as more options for a mixed task and data parallel execution are available. Table II lists the speedup of all scheduling algorithms averaged over all task graph sizes relative to the *Dataparallel* scheduler. The results of all other scheduling algorithms lie closer together for 16 available processors and range from a speedup of 1.89 (*Taskparallel*) to a speedup of 2.75 (*CPR*). As Figure 7 shows, *CPR* constantly achieves the best average quality for 16 available processors. On the other hand *CPR* gets outperformed by *MSAA* in 18%, by *CPA* in 12% and by *TSAS* in 4% of the test cases.

For 256 available processors the schedules with the lowest average makespan are delivered by *TSAS* with a speedup of 3.14 compared to a dataparallel execution, followed by *CPR* with a speedup of 2.8. The average results obtained by *MSAA* are located in the mid range and *CPA* exhibits an unusual behavior. The makespan of the schedules delivered by *CPA* increases if more processors are available for most scheduling problems. For 16 processors *CPA* could achieve competitive makespans but is worse than the *Taskparallel* scheduler for 256 processors.

Table III shows for each scheduling algorithm the number of times it could generate the schedule with the lowest makespan for different number of nodes. The left number corresponds to 16 available processors and the right number belongs to 256 available processors. The *Dataparallel* and *Taskparallel* schedulers never construct a minimal schedule. As the average results already showed, *CPR* obtains the best results for 16 processors and *TSAS* has the lead for 256 processors.

In summary the results state that there is no scheduling algorithm that clearly dominates the test field. Our results for low numbers of processors agree with the findings from [5], [4], where *CPR* achieves the best quality and *CPA* was shown to be competitive to other scheduling algorithms. We have shown that for a high number of processors the quality of *CPA* gets worse and *CPR* is mostly outperformed by *TSAS*. Especially the experiments in this paper show that the M-Task programming approach clearly outperforms a pure dataparallel execution and is therefore a suitable model for parallel computation. Table IV gives an overview of the suggested scheduling algorithm depending on the size of the input problem (number of nodes), the number of processors and the runtime of the scheduling algorithm.

V. Conclusion and Future Work

In this paper we have evaluated a variety of scheduling algorithms for M-Tasks with dependencies belonging to the class of *Allocation-and-Scheduling-based* algorithms. We compared the runtime of the scheduling algorithms and the makespans of the generated schedules. If the makespan of the resulting schedule should be minimized, *CPR* is a good choice for a low number of processors and the runtime is reasonably low in this case. For a high number of processors *TSAS* is faster and constructs better schedules compared to *CPR*. As a schedule for an application usually depends on the input data size and has therefore to be recomputed multiple times, the running time of the applied scheduling algorithm becomes an important issue. In this case *CPA* is a good choice for a low number of processors, but the resulting schedules are not competitive for a high number of processors. *MSAA* offers a good scalability for high numbers of nodes and processors and produces schedules with a middle-ranked makespan.

Future work includes the examination of additional categories of scheduling algorithms. It is planned to make the algorithms that are implemented in the toolkit available to other applications in form of a library.

References

[1] H. Bal and M. Haines, “Approaches for integrating task and data parallelism,” *IEEE Concurrency*, vol. 6, no. 3, pp. 74–84, 1998.

[2] J. Dümmler, R. Kunis, and G. Rünger, “A Scheduling Toolkit for Multiprocessortask-programming with Dependencies,” *submitted for publication*.

[3] R. Lepere, D. Trystram, and G. J. Woeginger, “Approximation algorithms for scheduling malleable tasks under precedence constraints,” *Lecture Notes in Computer Science*, vol. 2161/2001, 2001.

[4] A. Radulescu, C. Nicolescu, A. van Gemund, and P. Jonker, “CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems,” in *IPDPS ’01: Proc. of the 15th Int. Par. & Dist. Proc. Symp.* IEEE Computer Society, 2001, p. 39.

[5] A. Radulescu and A. van Gemund, “A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling,” in *Proc. of the 2001 Int. Conf. on Parallel Processing.* IEEE Computer Society, 2001, pp. 69–76.

[6] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, “A framework for exploiting task and data parallelism on distributed memory multicomputers,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 11, pp. 1098–1116, 1997.

[7] T. Rauber and G. Rünger, “Compiler support for task scheduling in hierarchical execution models,” *J. Syst. Archit.*, vol. 45, no. 6-7, pp. 483–503, 1998.

[8] ——, “A Transformation Approach to Derive Efficient Parallel Implementations,” *IEEE Trans. on Software Engineering*, vol. 26, no. 4, pp. 315–339, 2000.

[9] J. Valdes, R. E. Tarjan, and E. L. Lawler, “The recognition of series parallel digraphs,” Tech. Rep., 1979.