# Layer-Based Scheduling Algorithms for Multiprocessor-Tasks 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*

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. The scalability is impacted by the use of communication operations, e.g. broadcast operations, whose runtime exhibits a logarithmic or linear dependence on the number of utilized processors. The multiprocessor-task programming model can help to reduce the communication overhead, but requires an appropriate schedule for an efficient execution. Many heuristics and approximation algorithms are available for this scheduling task. The choice of a suitable scheduling algorithm is an important factor in the development of multiprocessor-task applications. In this paper, we consider *layer-based* scheduling algorithms and compare their runtimes for large task graphs consisting of up to 1000 nodes and target systems with up to 256 processors. Furthermore, we introduce an extension methodology to enable scheduling algorithms for independent multiprocessor-tasks to handle precedence constraints.

## 1 Introduction

Modular applications from scientific computing can be implemented using the multiprocessor-task (M-Task) programming model with precedence constraints, which has been shown to yield better results than a pure data parallel or a pure task parallel execution, especially for distributed memory platforms with a large number of processors. In the M-Task programming model, a parallel application is defined as a set of M-Tasks where each M-Task can be executed on an arbitrary subset of the available processors of the target machine. Dependencies arise from data and control dependencies between M-Tasks. Independent M-Tasks can be executed in parallel on disjoint subsets of the available processors.

The execution of an M-Task application is based on a schedule that assigns each M-Task a subset of the available processors and fixes the execution order of the M-Tasks. In general, for a given M-Task program many different schedules are 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. As determining the optimal schedule is an NP-hard problem, many scheduling heuristics and approximation algorithms have been proposed to get a near optimal solution to this problem.

We define two main classes of scheduling algorithms for M-Tasks with precedence constraints: *allocation-and-scheduling-based* and *layer-based* algorithms. Algorithms of these categories were implemented in a scheduling toolkit[1] that enables application developers to automatically determine the best algorithm for any combination of M-Task program and target hardware platform. In this paper, we examine *layer-based* scheduling

algorithms. *Allocation-and-scheduling-based* algorithms have been considered in [2]. Much theoretical and practical research has been done for scheduling sets of independent M-Tasks but most scientific applications can only be modeled with precedence constraints. Therefore, we present an extension strategy for the problem of scheduling tasks without precedence constraints to the scheduling problem with dependencies and apply this strategy to several scheduling algorithms. Our extension strategy enables the extension of any scheduling algorithm for independent M-Tasks to support precedence constraints.

The paper is structured as follows: Section 2 explains the M-Task programming model with dependencies. Section 3 outlines our extension methodology and gives an overview of the considered *layer-based* scheduling algorithms. The obtained benchmark results are discussed in Section 4. Section 5 concludes the paper.

## 2 Multiprocessor-Task programming model

In the M-Task programming model a parallel application is represented by an annotated directed acyclic graph (M-Task dag) $G = (V, E)$. An example of a small dag is given in Figure 1. A node $v \in V$ corresponds to the execution of an M-Task, which is a parallel program part implemented using an SPMD programming style. An M-Task 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 denoted as the allocation of the task $v$.

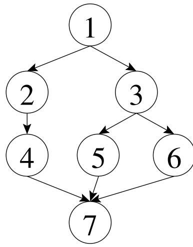

Figure 1. Example of a small M-Task dag.

A directed edge $e = (u, v) \in E$ represents precedence constraints between two M-Tasks $u$ and $v$, i.e. $u$ produces output data that $v$ requires as input data. Therefore, $u$ and $v$ have to be executed one after another. Edges may lead to a data re-distribution if the processor group changes, i.e. $g_u \neq g_v$ or if $u$ and $v$ 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 $T_v : [1, \ldots, P] \to \mathbb{R}^+$ and each edge $e = (u, v) \in E$ is assigned a communication cost $T_{comm}(u, v)$.

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.

$$\forall\, u, v \in V, (u, v) \in E \quad T_{S_u} + T_u(|g_u|) + T_{comm}(u, v) \leq T_{S_v}.$$

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

$$\forall\, u, v \in V \quad [T_{S_u}, T_{S_u} + T_u(|g_u|)] \,\cap\, [T_{S_v}, T_{S_v} + T_v(|g_v|)] \neq \emptyset \implies g_u \cap g_v = \emptyset$$

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 do not take data re-distribution costs into account. This is feasible because the communication costs in scientific applications are usually a magnitude lower compared to the computational costs of the M-Tasks. Furthermore, it is often possible to hide at least parts of these costs by overlapping of computation and communication.

## 3 Layer-based Scheduling Algorithms

There has been a lot of research regarding scheduling algorithms for independent M-Tasks. However, these scheduling algorithms cannot cope with precedence constraints between M-Tasks. This limitation can be avoided using layer-based scheduling algorithms[3] for M-Tasks with precedence constraints. These algorithms utilize a *shrinking phase* and a *layering phase* to decompose an M-Task dag into sets of independent M-Tasks, called layers. The subsequent *layer scheduling phase* computes a schedule for each layer in isolation. Additionally, we introduce an explicit fourth phase, the *assembling phase* constructing the output schedule for the M-Task dag. Our extension strategy enables the combination of the shrinking phase, the layering phase and the assembling phase with a scheduling algorithm for independent M-Tasks in the layer scheduling phase. Therefore, any scheduling algorithm for independent M-Tasks can be extended to support M-Task dags. The phases are illustrated in Figure 2 utilizing the small example M-Task dag from Figure 1.

This extension method has been applied to the following algorithms for independent M-Tasks described in Section 3.3: The scheduling algorithm with approximation factor two[4] was extended to the *Ext-Approx-2* algorithm. The *Ext-Dual-*$\sqrt{3}$ algorithm was derived from the dual-$\sqrt{3}$ approximation algorithm[5] and the dual-3/2 approximation algorithm[6] was transformed into the *Ext-Dual-*$3/2$ algorithm. A similar approach was utilized for malleable tasks[7]. This approach also executes phases to derive independent tasks, but does not include a shrinking phase and the malleable tasks consist of single processor tasks.

### 3.1 Shrinking Phase

The shrinking phase reduces the solution space of the scheduling problem by replacing subgraphs of the M-Task dag that should be executed on the same set of processors by a single node leading to a new dag $G' = (V', E'), |V'| \leq |V|, |E'| \leq |E|$. This phase is implemented as follows: At first we try to find all maximum linear chains in the dag $G$. A linear chain is a subgraph $G_c$ of $G$ with nodes $\{v_1, .., v_n\} \subset V$ and edges $\{e_1, .., e_{n-1}\} \subset E$, with $e_i = (v_i, v_{i+1}),\ i = 1, .., n-1$, and $v_i$ is the only predecessor of $v_{i+1}$ and $v_{i+1}$ is the only successor of $v_i$ in $G$. A maximum linear chain $G_{c_m}$ is a chain which is no subgraph of another linear chain. We then aggregate the nodes and edges of each $G_{c_m}$ to a new node $v_{c_m}$ having the sum of the computation costs of all nodes plus the sum of the communication costs of all edges as computation cost. This aggregation implies that each $G_{c_m}$ in $G$ is replaced by the aggregated node $v_{c_m}$ connecting all incoming edges of $v_1$ and all outgoing edges of $v_n$ of the maximum chain $G_{c_m}$ with the new node $v_{c_m}$.

### 3.2 Layering Phase

In the layering phase the nodes of the shrinked dag $G'$ are partitioned into $l$ disjoint subsets of nodes where each subset contains only independent nodes. Such a subset without precedence constraints is called a layer. In the following the nodes of a layer $i, i = 1, \ldots, l$,

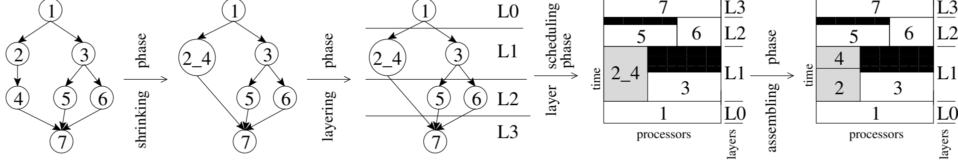


Figure 2. Illustration of the phases of the *layer-based* scheduling algorithms.

are denoted as $V_{L_i}, V_{L_i} \subseteq V'$ with $\bigcup_{i \in 1..l} V_{L_i} = V'$. Additionally, there has to exist an ordering between the layers $i$ and $k$, $i, k = 1, \ldots, l$ given by:

$$\forall u \in V_{L_i}, \forall v \in V_{L_k} \text{ if there is a path from } u \text{ to } v \text{ in } G' \implies i < k.$$

The scheduling decision in the next phase becomes more flexible when the number of layers gets smaller and the number of nodes in a layer gets larger. Therefore a greedy approach is used, which performs a breadth first search and puts as many nodes as possible into a layer, to realize this phase.

### 3.3 Layer Scheduling Phase

In this phase an M-Task schedule is computed for each constructed layer $V_{L_i}, i = 1, \ldots, l$ in isolation. In the following we omit the index $i$ and use $V_L$ for the layer to be scheduled.

**TwoL-Level** determines the total execution time for each possible partitioning of the set of available processors into $\kappa, \kappa = 1, \ldots, \min(P, |V_L|)$ subgroups $\hat{g}_{\kappa,1}, \ldots \hat{g}_{\kappa,\kappa}$ of about equal size[3]. The schedule for each of these partitionings is computed by adopting a list scheduling heuristic. In each step of this heuristic the M-Task $v \in V_L$ is assigned to group $\hat{g}^* \in \{\hat{g}_{\kappa,1}, \ldots \hat{g}_{\kappa,\kappa}\}$, where $\hat{g}^*$ is the first subgroup becoming available and $v$ is the M-Task with the largest execution time. The final processor groups $g_1, \ldots, g_{\kappa^*}$ are computed by a subsequent group adjustment step from the groups $\hat{g}_{\kappa^*,1}, \ldots, \hat{g}_{\kappa^*,\kappa^*}$, where $\kappa^*$ denotes the partitioning resulting in a minimum runtime.

**TwoL-Tree** starts by constructing a tree for each M-Task $v \in V_L$ consisting of a single node[8]. A dynamic programming approach is used to find all unordered pairs of trees $\{t_1, t_2\}$ with an equal depth and disjoint sets of M-Tasks. For each pair $\{t_1, t_2\}$ a new tree $t$ with a new root node and children $t_1$ and $t_2$ is created. Each tree represents a schedule of the contained M-Tasks. The inner nodes of the trees are annotated with a cost table containing the execution time of the whole subtree for all possible processor group sizes $gs = 1, \ldots, P$. A second annotation defines whether the schedules represented by the children of the node should be executed one after another or in parallel on disjoint processor groups. Finally, a set of trees each containing all nodes of the current layer is constructed, where each such tree symbolizes a different schedule. The output schedule of the algorithm is constructed from the tree which admits the minimum execution time.

**Approx-2** is a 2-approximation algorithm for a set of independent M-Tasks[4]. It partitions the M-Tasks of a layer into the sets $P(\tau)$ and $S(\tau)$. $P(\tau)$ contains the parallel M-Tasks that are assigned a number of processors such that their execution time is smaller than $\tau$ and

$S(\tau)$ is a set of M-Tasks that are assigned a single processor. The schedule is constructed by starting all M-Tasks of $P(\tau)$ at time 0 on disjoint processor groups and by using a list scheduling heuristic to assign the M-Tasks of $S(\tau)$ to the remaining processors. The optimal value of $\tau$ is selected by using binary search on an array of candidate values.

**Dual-$\sqrt{3}$** belongs to the class of dual approximation algorithms[5]. A dual-$\theta$ approximation scheduling algorithm takes a real number $d$ as an input and either delivers a schedule with a makespan of at most $\theta * d$ or outputs that there is no schedule with a makespan $\leq d$. Dual-$\sqrt{3}$ is a 2-shelve approximation algorithm that uses a canonical list algorithm or a knapsack solver to construct the schedule. At first the algorithm determines the minimal number of processors $p_v$ for each task $v \in V_L$ such that the execution time does not exceed $d$. If $\sum_{v \in V_L} p_v < P$ a canonical list algorithm is used to pack all tasks in the first shelve starting at time 0. Otherwise three subsets of the tasks are created. $S_1$ consists of the tasks with $T_v(p_v) > \lambda, \lambda = \sqrt{3}d - d$. In $S_2$ tasks with $\frac{1}{2}d \leq T_v(p_v) \leq \lambda$ and in $S_3$ tasks with $T_v(p_v) < \frac{1}{2}d$ are stored. $S_2$ and $S_3$ are packed into the second shelve that starts at time $d$. The first shelve is filled with tasks of $S_1$ until the sum of the needed processors gets larger than $P$. All remaining tasks of $S_1$ are packed into the second shelve with a processor number of $q_v > p_v$ determined by a knapsack algorithm.

**Dual-$3/2$** is a dual approximation algorithm with $\theta = \frac{3}{2}$ that constructs schedules consisting of two shelves[6]. The algorithm starts by removing the set $T_S$ of small M-Tasks from the layer and defining a knapsack problem to partition the remaining M-Tasks into the sets $T_1$ and $T_2$. The M-Tasks of $T_1$ and $T_2$ are assigned the minimal number of processors such that their execution time is less than $d$ or $\frac{d}{2}$, respectively. The initial schedule is obtained by first inserting the M-Tasks of $T_1$ at start time 0 in the first shelve, the M-Tasks of $T_2$ at start time $d$ in the second shelve and by assigning the small M-Tasks of $T_S$ to idle processors. The output schedule is calculated by repeatedly applying one of three possible transformations to the schedule until it becomes feasible. These transformations include moving tasks between sets and stacking tasks on top of each other.

### 3.4 Assembling Phase

This phase combines the layer schedules and inserts necessary data re-distribution operations between the layers resulting in a global schedule for the M-Task dag. The layer schedules are combined following the ordering of the layers defined in the layering phase. Furthermore, the shrinked nodes, aggregated nodes of maximum linear chains $G_{c_m}$, are expanded, i.e. all nodes $v \in V$ of a shrinked node $v_{c_m} \in V'$ are executed one after another on the same processor group. This is realized for each $G_{c_m}$ with nodes $\{v_1, ..v_n\} \subset V$ and edges $\{e_1, ..e_{n-1}\} \subset E$ as follows: Schedule node $v_1$ of the chain on processor group $g_{v_{c_m}}$ at time $T_{S_{v_{c_m}}}$. The following nodes $v_i, i = 2, \ldots, n$ of $G_{c_m}$ are scheduled on $g_{v_{c_m}}$ at time $T_{S_{v_{i-1}}} + T_{v_{i-1}}(|g_{v_{c_m}}|) + T_{comm}(v_{i-1}, v_i)$ including data re-distributions between two nodes $v_{i-1}$ and $v_i$.

## 4 Simulation Results

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. For each number of nodes we constructed a *test set* consisting of 100 different synthetic M-Task dags

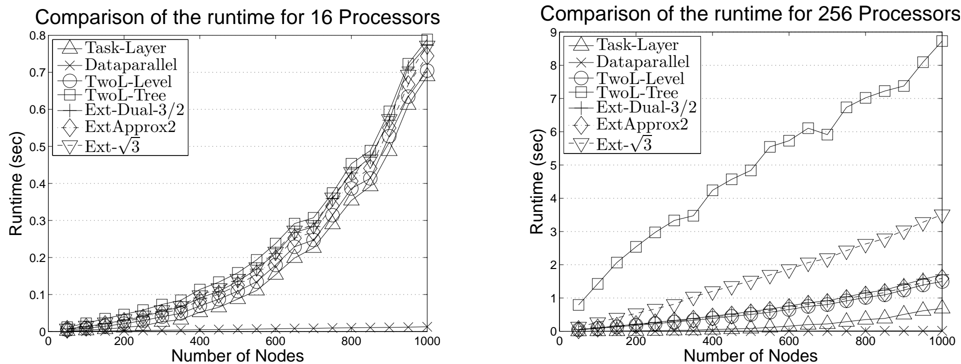


Figure 3. Comparison of the average runtime of the *layer-based* scheduling algorithms for task graphs with 50 to 1000 nodes and 16 (left) and 256 (right) available processors.

belonging to the class of series-parallel-graphs (SP-graphs). SP-graphs reflect the regular structure of most scientific applications. All nodes of the graphs 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 $T_{seq}(T_{seq} > 0)$.

First we present the runtime of the implemented scheduling algorithm averaged over each test set. For a comparison with a pure data or task parallel execution we use *Dataparallel* and *Task-Layer* respectively. *Task-Layer* uses a list scheduling algorithm in the layer scheduling phase that assigns each node of the shrinked dag one processor and schedules the tasks in parallel. Figure 3 shows the runtime of the scheduling algorithms for 16 (left) and 256 (right) processors depending on the number of nodes. In both cases *Dataparallel* achieves the lowest runtimes and *TwoL-Tree* achieves the highest runtimes. For 16 available processors the schedulers, except *Dataparallel*, achieve nearly the same runtime that is below one second for 1000 nodes and show a similar behavior if the number of nodes is increased. *Task-Layer* achieves the second lowest runtimes that are in average 20 times smaller than that of *Dataparallel*. All other algorithms have a nearly constant deviation from the runtimes of *Task-Layer* of at most 0.1 seconds. In the case of 256 available processors *TwoL-Tree* achieves the highest runtimes. The runtime is a factor of 3 to 9 higher compared to the other schedulers for an M-Task dag with 1000 nodes. *Ext-Dual-*$\sqrt{3}$ is the second slowest scheduler followed by *TwoL-Level*, *Ext-Approx-2* and *Ext-Dual*-$3/2$ that have significantly lower runtimes. *Dataparallel* and *Task-Layer* again achieve the lowest runtimes. The runtimes of the schedulers for 256 available processors increases linearly in the number of nodes of the dag. This behavior results from the linear runtimes of the shrinking phase, the layering phase, and the assembling phase, which have a worst case runtime of $\mathcal{O}(V + E)$. The layer scheduling phase is only executed for small subsets $V_L$ of the available nodes $V$ ($|V_L| \ll |V|$). Additionally, the runtime of the layer scheduling phase is dominated by the number of processors $P$, because $P > |V_L|$ for many layers.

Figure 4 shows the makespan of the produced schedules for 16 (left) and 256 (right) processors depending on the number of nodes. The results for 16 available processors show that *TwoL-Tree* and *TwoL-Level* produce the lowest makespans with a deviation of $1.8\%$. They are followed by the three adapted approximation algorithms *Ext-Dual*-$3/2$,

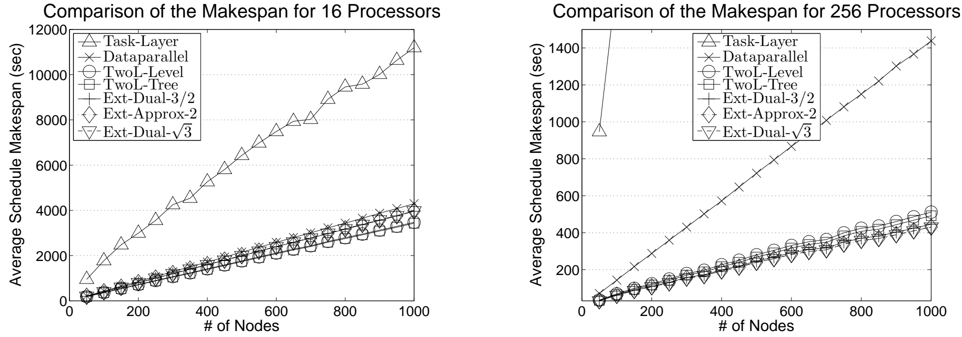


Figure 4. Comparison of the average makespan of the *layer-based* scheduling algorithms for task graphs with 50 to 1000 nodes and 16 (left) and 256 (right) available processors.

*Ext-Dual*-$\sqrt{3}$, and *Ext-Approx-2* with 14.5%, 14.7%, and 14.7% higher makespans. The makespan of the schedules produced by *Dataparallel* are also competitive (23.5% higher than the ones from *TwoL-Tree*). The worst schedules were produced by *Task-Layer* with a 281% higher average makespan than *TwoL-Tree*. The results show that the makespans of the produced schedules increase linearly in the number of nodes.

Considering the results for target machines with 256 processors, Figure 4 (right) shows that the two TwoL schedulers changed their places with the adapted approximation algorithms. The best average schedules were produced by *Ext-Dual*-$\sqrt{3}$ and *Ext-Approx-2* with a very small deviation, followed by *Ext-Dual*-3/2 with 2.8% higher makespans. *TwoL-Tree* and *TwoL-Level* have a 11.6% and 16.9% higher average makespan. The results show again that both, a pure data parallel and a pure task parallel execution are clearly outperformed by all other scheduling algorithms. The makespan of *Task-Layer* are in average 27 times smaller than the ones from *Ext-Dual*-$\sqrt{3}$ and *Ext-Approx-2* which produce the best schedules. *Dataparallel* creates schedules with an average makespan that is 3 times worse compared to *Ext-Dual*-$\sqrt{3}$. These results also show that the makespans of the produced schedules increase linearly in the number of nodes.

Table 1 shows the speedups of the produced schedules compared to *Dataparallel*. It shows that except *Task-Layer* all scheduling algorithms construct better schedules than *Dataparallel* on average. Also we have examined that the schedule with the lowest makespans were never created by *Task-Layer* or *Dataparallel*. The best schedules were

| Sched./Procs | 16 | 64 | 128 | 256 |
|---|---|---|---|---|
| Task Layer | 0.33 | 0.16 | 0.13 | 0.11 |
| TwoL-Level | 1.21 | 1.72 | 2.12 | 2.54 |
| TwoL-Tree | 1.24 | 1.80 | 2.24 | 2.66 |
| Ext-Dual3/2 | 1.08 | 1.73 | 2.31 | 2.90 |
| Ext-Approx-2 | 1.08 | 1.82 | 2.43 | 2.98 |
| Ext-Dual-$\sqrt{3}$ | 1.08 | 1.82 | 2.43 | 2.98 |

Table 1. Speedups of the produced schedules relative to the *Dataparallel* Scheduler.

| | low number of processors | high number of processors |
|---|---|---|
| low number of nodes | *TwoL-Tree** <br> *TwoL-Level*** | (*Ext-Dual*-3/2, *Ext-Dual*-$\sqrt{3}$ *Ext-Approx2*)*,** |
| high number of nodes | *TwoL-Tree** <br> *TwoL-Level*** | (*Ext-Dual*-$\sqrt{3}$, *Ext-Approx2*)* <br> (*Ext-Dual*-3/2, *Ext-Approx2*)** |

* best quality  ** good quality, reasonable runtime

Table 2. Recommended scheduling algorithms for different situations.

produced by *TwoL-Tree* for 16 available processors and by *Ext-Approx-2*, *Ext-Dual-*$\sqrt{3}$ or *Ext-Dual-*$3/2$ for 256 available processors. The obtained results of the runtimes of the algorithms and the quality of the schedules lead to the recommendations given in Table 2. The recommendations quote which algorithm should be utilized in which situation. A situation is determined by the attributes: size of the dag, number of available processors of the target machine, and the choice whether the best solution should be found or a good trade-off between quality and runtime should be reached.

## 5 Conclusion

In this paper we proposed an extension methodology for scheduling algorithms for independent M-Tasks to handle M-Tasks with precedence constraints, which is accomplished by a *layer-based* approach. Additionally, we presented a comparison of the extended algorithms with existing *layer-based* scheduling algorithms and derived a guideline, which algorithm developers should utilize depending on the parallel application and target platform. For this purpose we took the runtime of the scheduling algorithms as well as the quality of the generated schedules into account. The results show that a mixed task and data parallel execution derives better results than a pure data and task parallel execution in all considered cases. Especially for a large number of processors, the M-Task approach results in much higher speedups.

## References


1. J. Dümmler, R. Kunis, and G. Rünger. A Scheduling Toolkit for Multiprocessortask Programming with Dependencies. In *Proc. of the 13th Int. European Conf. on Par. and Distr. Comp. (Euro-Par 07)*, volume 4641 of *LNCS*, Rennes, France, 2007.
2. J. Dümmler, R. Kunis, and G. Rünger. A comparison of scheduling algorithms for multiprocessortasks with precedence constraints. In *Proc. of the 2007 High Performance Computing & Simulation (HPCS'07) Conference*, pages 663–669, 2007.
3. T. Rauber and G. Rünger. Compiler support for task scheduling in hierarchical execution models. *J. Syst. Archit.*, 45(6-7):483–503, 1998.
4. W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *SODA '94: Proc. of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 167–176. Society for Industrial and Applied Mathematics, 1994.
5. G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *SPAA '99: Proc. of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 23–32. ACM Press, 1999.
6. G. Mounie, C. Rapine, and D. Trystram. A $\frac{3}{2}$-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks. *SIAM J. on Computing*, 37(2):401–412, 2007.
7. W. Zimmermann and W. Löwe. Foundations for the integration of scheduling techniques into compilers for parallel languages. *Int. J. of Comp. Science and Engineering*, 1(3/4), 2005.
8. T. Rauber and G. Rünger. Scheduling of data parallel modules for scientific computing. In *Proc. of the 9th SIAM Conf. on Parallel Processing for Scientific Computing (PPSC), SIAM(CD-ROM)*, San Antonio, Texas, USA, 1999.