CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–17
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

One Step towards Bridging the Gap between Theory and Practice in Moldable Task Scheduling with Precedence Constraints

Sascha Hunold*

Research Group Parallel Computing, Institute of Information Systems, Vienna University of Technology, Austria

SUMMARY

Due to the increasing number of cores of current parallel machines and the growing need for a concurrent execution of tasks, the problem of parallel task scheduling is more relevant than ever, especially under the moldable task model, in which tasks are allocated to a fixed number of processors before execution. Much research has been conducted to develop efficient scheduling algorithms for moldable tasks, both in theory and practice. The problem is that theoretical as well as practical approaches expose shortcomings, e.g., many approximation algorithms only guarantee bounds under assumptions, which are unrealistic in practice, or most heuristics have not been rigorously compared to competing approximation algorithms. In particular, it is often assumed that the speedup function of moldable tasks is either non-decreasing, sub-linear or concave. In practice, however, the resulting speedup of parallel programs on current hardware with deep memory hierarchies is most often neither non-decreasing nor concave.
We present a new algorithm for the problem of scheduling moldable tasks with precedence constraints for the makespan objective and for arbitrary speedup functions. We show through simulation that the algorithm not only creates competitive schedules for moldable tasks with arbitrary speedup functions, but also outperforms other published heuristics and approximation algorithms for non-decreasing speedup functions. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: multiprocessor task scheduling, parallel and moldable tasks, makespan optimization, linear programming, arbitrary speedup functions

1. INTRODUCTION

The problem of scheduling parallel tasks has been intensively studied in the past, while it originally stems from the need of improving the utilization of massively parallel processors (MPPs) [1]. In this context, a parallel task can be executed by more than one processor, and researchers have attempted to understand the implications of different job scheduling strategies on the utilization of parallel systems both in theory and practice. Drozdowski distinguishes between three *types of parallel tasks* [2] (called *job flexibility* by Feitelson et al. [1]): (1) the *rigid task* requires a predefined fixed number of processors for execution, (2) the *moldable* task for which the number of processors is decidable before the execution starts, but stays unchanged afterwards, and (3) the *malleable* task, where the number of processors may vary during execution.

In this work, we focus on the *moldable* task model for mostly practical reasons, since the malleable model would require additional effort from programmers, e.g., to redistribute data or synchronize thread groups whenever the number of assigned resources changes. Parallel applications using Pthreads or OpenMP are typical examples of moldable tasks for which users can

*Correspondence to: Sascha Hunold, Research Group Parallel Computing, Faculty of Informatics, Vienna University of Technology, Favoritenstrasse 16/184-5, A-1040 Vienna, Austria. E-mail: hunold@par.tuwien.ac.at

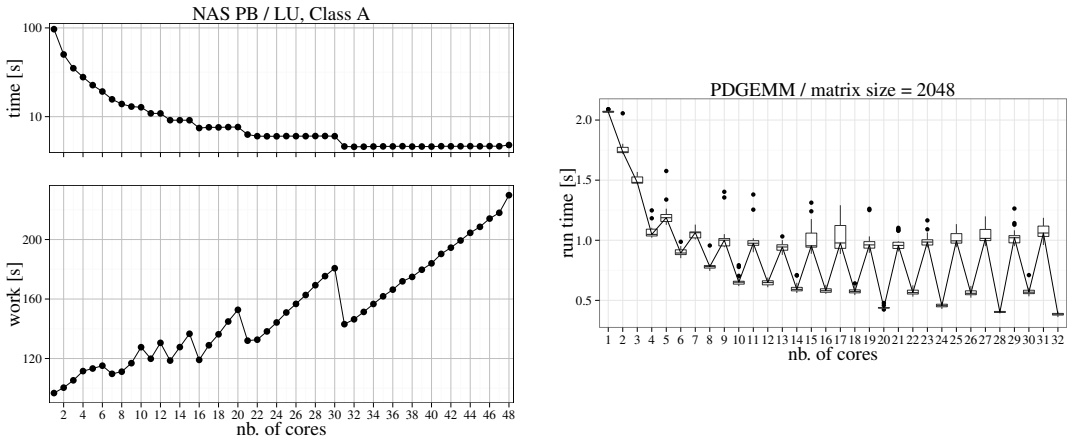


Figure 1. Left: run-time (top) and work (bottom) of LU benchmark on a shared memory machine (4 sockets, 48 cores, AMD Opteron 6168); Right: distribution of execution times of PDGEMM on a distributed memory machine (1 MPI process per node, 20 runs per core count, GBit Ethernet, AMD Opteron 6134).

specify the number of threads before the execution of a parallel program, while MPI applications are examples for distributed memory machines.

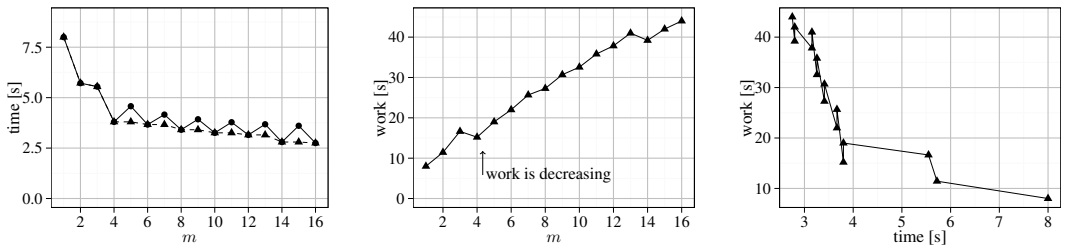
Today, researchers in parallel computing face the question of how to efficiently program the available processors (or cores). A well-known approach is to model an application as *directed cyclic graphs (DAGs)*, where edges make data dependencies explicit and nodes represent computations. The MAGMA library is an example, where DAGs represent parallel applications [3].

Now we want to turn to the problem of executing a DAG of tasks, in which each node represents a moldable task. For such DAGs, a scheduling algorithm has to determine (1) the next task to be executed and (2) the set of processors to which a task is allocated. In the scheduling literature, this is known as *scheduling problem for moldable tasks with precedence constraints* [2] (sometimes also called malleable task scheduling [4, 5, 6, 7]). A common assumption is that the run-time function of each parallel task is non-increasing and the corresponding work function is non-decreasing in the number of processors, where the work is defined as the product of run-time and number of processors. Figure 1 shows two examples where this assumption is violated in practice. The two plots on the left show run-time and work of the NAS LU benchmark on a 48-core shared-memory machine (median of five runs). The run-time is almost non-increasing but the corresponding work decreases several times, e.g., with 21 or 31 cores. The chart on the right shows the run-time of PDGEMM from ScaLAPACK (using GotoBLAS) on a distributed memory machine. Since this matrix multiplication routine works best on a square processor grid, we see an increased run-time for 11 or 13 processors. This "zigzagging" was already observed by van de Geijn and Watts [8].

We can say that the run-time of moldable tasks does not always decrease in practice when the number of assigned processors increases. This non-monotonous behavior of the run-time function has also consequence for the corresponding work function. Figure 2(a) shows an artificially created run-time function of some parallel task to exemplify the problem. Let us make the run-time function $p(k)$, where k denotes the number of processors, non-increasing by using the run-time $p(i-1)$ instead of $p(i)$ whenever $p(i-1) < p(i)$, i.e., the run-time function shown in Figure 2(a) is transformed into a non-increasing step function. The problem now is that the resulting work function, presented in Figure 2(b), is clearly not monotonically increasing. For example, the work function of this imaginary parallel task decreases at 4 and 14 processors. Similarly, if we plot the run-time and the corresponding work of the parallel task for all possible allocations, we see that the resulting function is not convex. Hence, from a practical point of view, the assumption of having a non-increasing run-time and a non-decreasing work hardly holds true on real machines.

To overcome the problem of choosing an algorithm for which the assumptions are not violated in practice, we propose an *algorithm for scheduling non-preemptive moldable tasks with precedence*

DOI: 10.1002/cpe



(a) (b) (c)

Figure 2. Possible problems with run-time and work functions when applying a simple strategy to force non-increasing run-time behavior. (a) original non-monotonous run-time function (solid line) and its non-increasing equivalent (dashed line); (b) resulting work function can be decreasing; (c) resulting work function is not convex in the processing time.

constraints for (1) *non-increasing run-time and non-decreasing work functions* and also (2) *arbitrary run-time and work functions*. In the three-field notation of Graham et al. [9], we investigate $P\,|\,any, NdSub, prec\,|\,C_{\max}$ and $P\,|\,any, prec\,|\,C_{\max}$, where P denotes a system with identical processors, any denotes the moldable task model, $NdSub$ denotes a nondecreasing sublinear speedup, and $prec$ represents the precedence constraints[†]. Our objective is to minimize the makespan $C_{\max}$.

We make several contributions to the field of moldable task scheduling. First, we propose a *new algorithm* (CPA13)[‡] that *supports arbitrary run-time functions* of moldable tasks. We show through simulation that our algorithm is competitive in a wide range of cases considered. Second, we compare schedules produced by CPA13 with schedules produced by several *approximation algorithms with performance guarantees*. To the best of our knowledge, this is the first experimental study that evaluates both CPA-style algorithms and approximation algorithms through simulation. We show that our *new algorithm* CPA13 produces *shorter schedules* even if *the run-time function of each parallel task is non-increasing*. Previous studies of scheduling heuristics for moldable tasks under precedence constraints use the absolute makespan as a metric to compare different heuristics. However, such analysis provides little evidence that the schedules produced by heuristics are not far away from the optimum. Hence, our third contribution is a rigorous comparison of algorithms, analyzing their *performance ratio*, which is defined as the *ratio of makespan to lower bound*.

Section 2 introduces the notation used in this article and Section 3 discusses related work. The new scheduling algorithm is introduced in Section 4, while Section 5 presents the simulation results before we conclude in Section 6.

2. NOTATION

We consider the problem of scheduling n moldable tasks with precedence constraints on m identical processors for the makespan objective $C_{\max}$. We study the offline and clairvoyant version of the problem, i.e., the entire DAG and the processing times for each node are known to the scheduling algorithm. The application is represented as a directed acyclic graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ denotes the set of moldable tasks and $E \subseteq V \times V$ represents the set of edges (arcs) between tasks ($e = |E|$). For every task v_j, $p_j(i)$ denotes the execution time of task j on i processors, and $w_j(i) = i \cdot p_j(i)$ denotes its work. Further, variable α_j refers to the number of processors allotted to task j.

The functions $p_j(i)$ and $w_j(i)$ are often assumed to be monotonic [10, Chap. 26], i.e., $p_j(i)$ is non-increasing and $w_j(i)$ non-decreasing in i. Formally, $p_j(i) \geq p_j(k)$ and $w_j(i) \leq w_j(k)$, for $i \leq k$.

[†] for more details on notation see [2, 9, 10]
[‡] Critical Path and Area-based Scheduling (CPA), "13" refers to the original year of working on this algorithm

DOI: 10.1002/cpe

Some algorithms require that $p_j(i)$ not only needs to be non-increasing, but also *convex* in the interval $[1, m]$. The work W of a DAG is computed as $W(\alpha) = \sum_{v_j} p_j(\alpha_j)\alpha_j$, where α denotes the allotment vector.

3. RELATED WORK

The problem of scheduling rigid tasks, where precedence constraints are given as a set of chains $P2|size_j, chain|C_{\max}$ is strongly NP-hard for $m \geq 2$ [10]. For the more general problem of scheduling moldable tasks with precedence constraints several approximation algorithms have been proposed. Lepère et al. presented an approximation algorithm with performance guarantee of $3 + \sqrt{5} \approx 5.236$ [4], where the scheduling problem is decomposed into an allotment and a mapping problem. The allotment problem—determining the number of processors for each moldable task—is solved by applying Skutella's method for obtaining an approximate solution to the discrete time-cost trade-off problem [11]. Jansen and Zhang improved the approximation ratio (to ≈ 4.73) by changing the rounding strategy for the fractional solution produced by the linear relaxation of the discrete problem [5]. All algorithms presented in [4, 5] require monotony of run-time and work, and the most recent algorithm by Jansen and Zhang (with approximation ratio ≈ 3.29) additionally requires that "the work function of any malleable task is non-decreasing in the number of processors and is convex in the processing time" [6]. In 2013, Chen and Chu reformulated the original algorithm of Jansen and Zhang [5] under the assumptions that the processing time is non-increasing in the number of processors and the work function is convex in the processing time [12]. Their proposed algorithm obtained an approximation ratio of $2 + \sqrt{2} \approx 3.4142$, and 2.9549 for the case that the processing time is strictly decreasing in the number of processors. To obtain these bounds, Chen and Chu pre-compute all possible virtual tasks and the corresponding virtual work for all moldable tasks. The knowledge of execution time and work of all virtual tasks allows the authors to modify the linear program, so that the obtained results have the right properties to allow an improved rounding.

Radulescu and van Gemund exploited similar properties of the problem as done by Jansen and Zhang or Lepère et al. [4, 5] for solving the allotment problem, and thus, their Critical Path and Area-based Scheduling (CPA) algorithm is based on the fact that the average work W/m and the length of the critical path L are lower bounds of $C_{\max}$ [13]. Their CPA algorithm starts by allocating a single processor to each task, then inspects all tasks on the critical path, and adds one processor to task j that reduces the length of the critical path by maximizing the following expression $\left(\frac{p_j(i)}{i} - \frac{p_j(i+1)}{(i+1)}\right)$. The allocation process repeats until the critical path L is smaller than the average work (W/m). Bansal et al. discovered that CPA should take task parallelism better into account [14]. More precisely, the allocation routine of CPA often produces large processor allotments, with the consequence that tasks—that could have potentially been executed in parallel—need to run sequentially. Bansal et al. introduced the Modified CPA (MCPA) algorithm, which considers the depth of tasks in the allocation phase. In particular, a task is not allotted more processors if already m processors have been allotted to (critical) tasks in the same depth. We showed in previous work how low-cost improvements to MCPA, such as relaxing the allotment constraints per precedence level or allowing allocation reductions, can improve the performance (average makespan) significantly [15].

Desprez and Suter proposed an algorithm to optimize both the makespan and the total work when scheduling a DAG [16]. They introduced the bi-criteria optimization algorithm (BiCPA) that computes the processor allotment for m different cluster sizes $[1, 2, \ldots, m]$ and selects the allotment that optimizes a given makespan-work ratio. BiCPA produces "short" and "narrow" schedules, but having the disadvantage of increasing the computational complexity significantly.

All algorithms described above assume non-increasing run-time and non-decreasing work functions. For the case of arbitrary run-time functions, only few algorithms have been proposed so far. Günther et al. presented a fully polynomial approximation scheme (FPTAS) for this scheduling problem [7]. As the general problem is strongly NP-hard, they looked at DAGs with bounded width

DOI: 10.1002/cpe

and developed a dynamic program to compute the solution. For practical applicability, the FPTAS has a rather large complexity of $O((\frac{n^3}{\epsilon})^{2\omega}m^{2\omega})$, where ω denotes the maximum width of a DAG.

In previous work, we already addressed the challenge of dealing with arbitrary run-time functions by using an evolutionary algorithm (EA) to find short schedules [17]. The proposed algorithm EMTS takes allotment solutions of several heuristics, like CPA and MCPA, and optimizes them using an $(\mu + \lambda)$-EA, where μ and λ denote the number of parents and offspring, respectively. The advantage of the meta-heuristic EMTS is that it finds short schedules, but it also has the disadvantage that the fitness evaluation of the offspring population is time-consuming.

In sum, efficient heuristics and approximation algorithms only exist for non-increasing run-time and non-decreasing work functions, and other known algorithms without such limitations (e.g., the FPTAS by Günther et al.) have high computational demands.

The problem of scheduling tasks under the moldable and malleable model has also practical relevance. Roderus et al. describe a case in which, as part of a larger computational problem, several sub-problems are solved independently in parallel, where one sub-problem consists of finding "all eigenvalues and eigenvectors of a set of symmetric matrices of different size" [18]. The problem here is that the tasks are relatively coarse-grained, and thus executing the tasks concurrently, each task on a single processor, would result in heavy load imbalance. To reduce the time needed to execute all tasks, they apply the moldable execution model for each task (although the authors call it malleable). Internally, tasks are implemented with LAPACK and ScaLAPACK routines, and thus the scheduling algorithm for moldable tasks employed needs to determine the number of processors for calling the routine `pdsygvx` from ScaLAPACK (eigenvalue computation). The authors experience two problems in practice that make their work very relevant for our contribution: (1) how to predict the execution time of moldable tasks depending on the number of processors, and (2) how to handle execution time functions that are not monotonically decreasing with the number of processors (as seen for PDGEMM, cf. Figure 1). The first problem is solved through profiling and curve-fitting and the second by making the execution time function of task t_i monotonically decreasing in the number of processors P by setting: $t_{i,p} = \min_{P}\{t_{P,S_i} : P \in \mathcal{P} \wedge P \leq p\}$.

Another recent work shows the applicability of the malleable task model to schedule dependent MapReduce jobs on parallel machines, for which Nagarajan et al. presented the scheduling algorithm FlowFlex [19]. The idea is that MapReduce jobs that consist of many single-processor jobs can be treated on a coarser level as malleable tasks, and it is possible to add or remove resources (machines) during execution. The authors propose several approximation algorithms for scheduling malleable tasks with precedence constraints, which can optimize either total weighted completion time or maximum weighted completion time.

4. A SCHEDULING ALGORITHM FOR ARBITRARY RUN-TIME FUNCTIONS

Our proposed scheduling algorithm applies concepts of the algorithms discussed before, e.g., reducing the critical path while keeping the overall work small. The main structure of our algorithm is based on CPA [13]. However, as it has been addressed in previous articles, CPA tends to allot too many processors to tasks even though the relative speedup is small. We solve this problem of limited task-parallelism by introducing the *relative run-time threshold* G, which defines the minimum run-time improvement that a larger allotment needs to provide to be considered as a possible solution. As shown later, this threshold is key for short and compact schedules. Our second objective is to conserve task parallelism as much as possible. A strategy to exploit task parallelism is employed by MCPA, which sums the processors that are assigned to all visited nodes in a certain DAG level, but may unmark once visited nodes. Only visited nodes are considered in the computation of the number of processors assigned to a DAG level. In contrast to MCPA, our algorithm considers all allotments of nodes that were once marked. As a third improvement, we optimize the mapping function, which selects processors for ready tasks. Since we have shown in a previous study that reducing processor allotments in the mapping step can significantly improve the overall schedule [15], CPA13 applies a binary search strategy on the task allocations to find, at a given time and for the current task, the

DOI: 10.1002/cpe

Table I. Overview of notation used.

m	–	number of processors	n	–	number of tasks (nodes)
e	–	number of edges ($\|E\|$)	L	–	length of critical path
W	–	overall work of DAG	G	–	min.rel. run-time improvement
v_j	–	task j	α_j	–	processors allotted to task v_j
$p_j(i)$	–	run-time of task v_j with i processors	b_j	–	benefit of task v_j
r_j	–	rel. run-time improvement	l_j^{bot}	–	bottom level of task v_j
l_j^{prec}	–	precedence level of task v_j	$\tilde{m}_d$	–	nb. processors in prec. level d
$allot_j$	–	list of possible allocations for v_j			

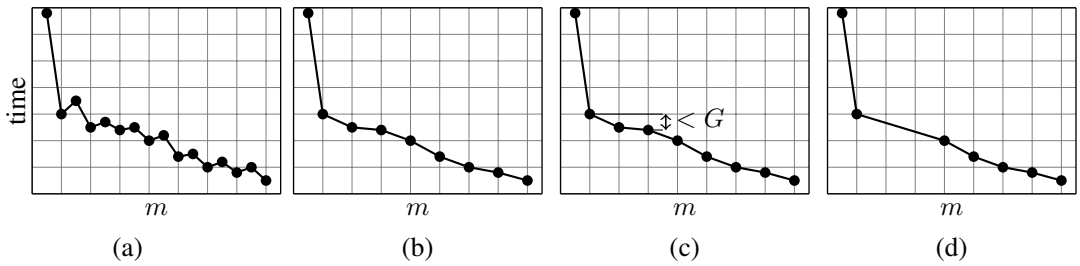


Figure 3. CPA13: steps when reducing the number of possible allocations for arbitrary run-time functions. (a) original non-monotonic run-time function; (b) remove all allocations for which the run-time is increasing; (c) remove allocations that do not provide enough speedup; (d) resulting run-time function used in CPA13.

smallest number of processors such that (1) the task can start earlier, (2) the completion time of the task is not delayed, and (3) the number of resources (processors) required is reduced.

4.1. Pseudocode

Algorithm 1 presents the allotment function of our algorithm named CPA13. For an easier comprehension we summarize the notation in Table I. Let us highlight the main steps of the algorithm. In the initialization phase, each task is allotted one processor and marked unvisited. We also pre-compute the possible benefit and the relative execution time reduction of each processor allotment (line 6–11), which is depicted in Figure 3. First, for each task we remove all processor allocations that would increase the execution time when adding more processors (Figure 3(b)). Second, if a larger allocation fails to provide the required relative run-time improvement of G, it will also be dismissed (Figure 3(c)). The resulting run-time model for each task is strictly decreasing in the number of processors as illustrated by the rightmost graphic in Figure 3.

In the second phase (line 12), all tasks on the critical path are collected, and for each task the benefit of the next larger possible allocation is evaluated, unless adding more processors to the precedence level of that task would exceed the total number of processors in the system (lines 21–22) (The precedence level denotes the shortest path from the root to a node without taking the computation time of tasks into account). After the best task has been found and its allotment has been increased, the values of the critical path L and the current work W have to be recomputed. The allotment process repeats until either the critical path L is smaller than (or equal to) W/m or no task satisfies the precedence level constraint.

Algorithm 2 presents the mapping procedure of CPA13, which first considers all ready tasks and extracts the task with highest priority. We use the highest bottom level as priority, i.e., the longest path from a node to the sink of the DAG. After extracting the ready task v_j, the procedure selects the α_j processors that first become idle. However, it might be possible to decrease the size of the allotment of v_j without increasing its completion time. In order to find such a smaller processor allotment for v_j we perform a binary search on v_j's allotments.

DOI: 10.1002/cpe

Algorithm 1 CPA13 allocation procedure

1: **for all** $v_j \in V$ **do**
2: $\alpha_j \leftarrow 1$
3: mark v_j as UNVISITED
4: $A_j \leftarrow$ list of increasing allotment sizes s.t. $\forall a_i, a_k \in A_j, i < k, a_i < a_k : p_j(a_i) > p_j(a_k)$
5: $\tilde{k} \leftarrow 1$
6: **for** k in $2 \ldots |A_j|$ **do**
7: $b_{j_k} \leftarrow \left(\frac{p_j(a_{\tilde{k}})}{a_{\tilde{k}}} - \frac{p_j(a_k)}{a_k} \right)$
8: $r_{j_k} \leftarrow \frac{p_j(a_{\tilde{k}}) - p_j(a_k)}{p_j(a_{\tilde{k}})}$
9: **if** $r_{j_k} \geq G$ **then**
10: $\tilde{k} \leftarrow k$
11: store (a_k, b_{j_k}) in list $allot_j$ of possible allotments for task v_j
12: **while** $L > W/m$ **do**
13: $V_{CP} \leftarrow$ collect tasks on critical path
14: $(v_b, \tilde{\alpha_b}, b_b) \leftarrow$ (**nil**, m, 0.0) // initialize current best temporary values
15: **for all** $d \in \{l_j^{prec} | v_j \in V\}$ **do**
// sum the number of processors assigned for visited tasks in the same layer
16: $\tilde{m}_d \leftarrow \sum_{v_l \in \tilde{V}} \alpha_l$ **where** $\tilde{V} = \{v_k \in V \text{ s.t. } l_k^{prec} = d \wedge v_k \text{ marked VISITED } \}$
17: **for all** $v_j \in V_{CP}$ **do**
18: $a_{j_k}, b_{j_k} \leftarrow$ size and benefit of task v_j's next larger allocation from $allot_j$
19: $s \leftarrow a_{j_k} - \alpha_j$ // absolute increase in number of processors
20: $d \leftarrow l_j^{prec}$
21: **if** $\tilde{m}_d + s \leq m \wedge b_{j_k} > b_b$ **then**
22: $(v_b, \tilde{\alpha_b}, b_b) \leftarrow (v_j, a_{j_k}, b_{j_k})$ // current best
23: **if** $v_b \neq$ **nil then**
24: $\alpha_b \leftarrow \tilde{\alpha_b}$ // increase allotment of task v_b
25: mark v_b as VISITED
26: recompute L and W
27: **else**
28: **break** // terminate while loop

Algorithm 2 CPA13 mapping procedure

1: **while not** all tasks scheduled **do**
2: $v_j \leftarrow$ find ready task with maximum l_j^{bot}
3: LET $C = \tau + p_j(\alpha_j)$ be the completion time if v_j were allocated α_j processors that become available first at time τ
4: $a_j \leftarrow$ use binary search on $allot_j$ to find the smallest allocation $a_j \leq \alpha_j$ for v_j s.t. $\tau_i + p_j(a_i) \leq C$ where $\tau_i \leq \tau$
5: schedule v_j onto first a_i processors that become available at time τ_i

4.2. Asymptotic Run-time Analysis

We determine the run-time complexity of CPA13 (the number of operations to perform) by examining the allocation and the mapping step separately. In the allocation phase of CPA13 (Algorithm 1), the benefit of a processor allotment is computed for all tasks ($O(nm)$). The body of the loop (line 12) determines the number of processors per precedence level ($O(n)$) and the critical path ($O(n+e)$). After selecting and modifying the best task, the critical path needs to be updated in $O(n+e)$ operations. The outer loop (line 12) will be executed at most $n \cdot m$ times since each of the n tasks can have a maximum of m processors allotted to it. Thus, the complexity of the allocation phase is $O(nm(n+e))$.

DOI: 10.1002/cpe

Table II. Summary of complexity results, "t.p." stands for "this paper".

algorithm	allocation procedure		mapping procedure	
CPA	[13]	$O(nm(n+e))$	[13]	$O(n\log n + nm + e)$
MCPA	[14]	$O(nm(n+e))$	[13]	$O(n\log n + nm + e)$
BiCPA-S	[16]	$O(nm(n+e))$	[16]	$O(m(n\log n + nm + e))$
JZ06	[5]	$O(LP(mn, n^2+mn))$	[4]	$O(mn+e)$
JZ12	[6]	$O(LP(mn, n^2+mn))$	[4]	$O(mn+e)$
CHEN13	[12]	$O(LP(mn, n^2+mn)+mn)$	[4]	$O(mn+e)$
CPA+NM+R	t.p.	$O(nm(n+e))$	[13]	$O(n\log n + nm + e)$
MCPA+NM+R	t.p.	$O(nm(n+e))$	[13]	$O(n\log n + nm + e)$
EMTS	[17]	input dependent	[13]	$O(r(n\log n + nm + e))$
CPA13	t.p.	$O(nm(n+e))$	[15]	$O(n(\log n + m\log m)+e)$

The mapping procedure (Algorithm 2) first extracts the task with the highest priority ($O(\log n)$ using a heap) and selects the processors that become idle next ($O(m)$). We apply a binary search ($O(\log m)$) on the processors, but which need to be sorted by increasing finishing time first ($O(m\log m)$). After a task was mapped to a set of processors, the algorithm visits every outgoing edge to detect new ready tasks and marks these edges to avoid re-evaluating them in later iterations. In total over all iterations, $O(e)$ edges are visited in the procedure. Given that the loop in line 1 runs once for every task, the overall complexity of the mapping procedure is $O(n(\log n + m\log m)+e)$.

Additionally, we present the asymptotic run-times of both the allocation and mapping procedure of related algorithms in Table II. JZ06 and JZ12 denote the algorithms of Jansen and Zhang from 2006 [5] and 2012 [6], where $LP(p,q)$ denotes "the time to solve a linear program with p variables and q constraints" [5]. For JZ06 the authors stated that the LIST scheduling function requires $O(nm)$ operations. As the number of edges may be greater than mn, we updated this run-time to $O(mn+e)$. The suffix "NM+R" identifies our modified versions of CPA and MCPA, which are aware of possibly increasing run-time functions (discussed in Section 5.4.3). The evolutionary algorithm EMTS is input-dependent since it takes as input solutions from other heuristics to obtain an initial population, and its run-time grows with the number of individuals produced in the optimization process. Thus, EMTS calls the mapping function for each individual, and r denotes the total number of individuals created.

5. EVALUATION

Our evaluation of CPA13 and the competing scheduling algorithms has been done through simulation for mainly two reasons: (1) simulation allows us to obtain a statistically significant number of results, and (2) not many truly moldable applications exist, which would limit the variety of experiments. Of course, an experimental study on real hardware with existing applications would be interesting. However, our goal is to conduct a first study that compares approximation algorithms and heuristics in this scheduling context. Thus, it is important to cover a variety of problem instances, e.g. different types of DAGs and platforms. For achieving this goal, a simulation study seems to be a good choice.

5.1. DAGs and Platforms

We consider two types of DAGs in the simulation: (1) application DAGs that mimic existing parallel algorithms and (2) synthetic (randomly generated) DAGs. The matrix multiplication algorithm by Strassen and the Fast Fourier Transformation (FFT) are our examples for *application-oriented DAGs*. To obtain different computation and scalability ratios, we keep the shape of these FFT and Strassen DAGs fixed but change the number of operations that each task needs to perform. The *synthetic (random) DAGs* are generated with DAGGEN [20] and contain 25, 50 or 100 nodes. Four parameters influence the DAG generation process: the *width* controls how many tasks can run in parallel, the *regularity* defines the uniformity of the number of tasks per DAG level, the *density*

DOI: 10.1002/cpe

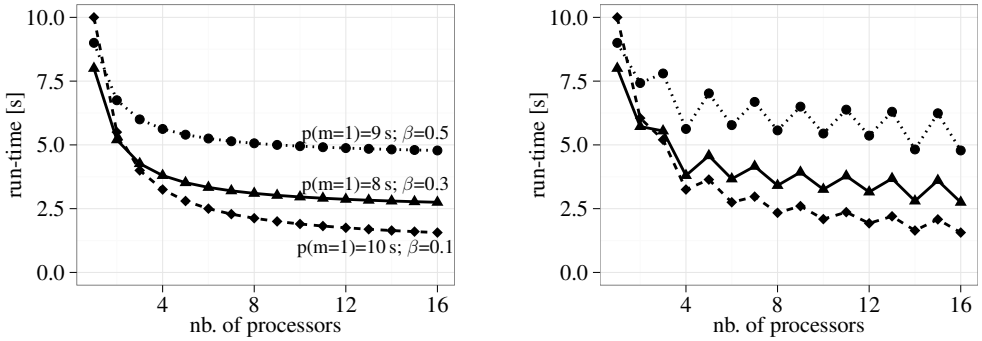


Figure 4. Examples of run-time models for moldable tasks: (left) *model 1*: non-increasing, monotonous, and convex execution time; (right) *model 2*: non-monotonous execution time.

specifies the number of edges, and the parameter *jump* controls if and how many DAG levels an edge (arc) may span.

The number of operations per task depends on a data size d (number of elements) and a function applied to the data, which were both randomly selected. The function $f(d)$ that is applied to the d elements defines the number of operations and is one of the following: stencil – d, sorting – $d \log d$, matrix multiplication – $(\sqrt{d})^3$. Function $f(d)$ and data size d only define the sequential time of a task. To obtain the parallel run-time of a moldable task, we apply Amdahl's law, which simply states that the speedup of a parallel application is limited by the fraction of sequential code. Thus, to obtain the speedup of a moldable task using Amdahl's law, we pick the non-parallelizable (sequential) fraction β (discussed below) of $f(d)$ randomly from a uniform distribution between 0 and 0.25.

For the purpose of better comparability, we used the same set of DAGs that had been used in the experiments in [16, 17]. The Cartesian product of the following values was used as parameter set to generate the irregular DAGs: *width* $= \{0.2, 0.5, 0.8\}$, *regularity* $= \{0.2, 0.8\}$, *density* $= \{0.2, 0.8\}$, and *jump* $= \{1, 2, 4\}$. In total, we created 400 FFT, 100 Strassen, 108 layered and 324 irregular DAGs ($n \times$ *width* $\times$ *regularity* $\times$ *density* $\times$ *jump*). *Layered DAGs* have edges only between adjacent precedence levels (*jump*=0) and the tasks in one tree level have an equal number of operations.

The platform model has two parameters: (1) the number of processors m and (2) the speed of the processors (in GFLOPS). We use two machine models in the simulations, where the first represents a Grid'5000[§] cluster (Grelon) with $m = 120$ processors providing 3.1 GFLOPS each (obtained with HP-LinPACK). The other machine model consists of $m = 48$ processors running at 6.7 GFLOPS (measured with GotoBLAS2), representing a shared-memory system at Vienna University of Technology.

We apply two run-time models to the parallel tasks in our simulation, which are described below.

Run-time Model 1: Since each task in the DAG generation process is assigned (1) a number of operations to perform and (2) the fraction of non-parallelizable code, we can base our run-time model on Amdahl's law. Let $p_i(1)$ be the sequential run-time of task v_i, determined by the ratio of the number of operations to be performed and the speed of the processor. Let β, $0 \leq \beta \leq 1$, be the non-parallelizable fraction of a parallel task, then the run-time of task v_i on k processors is given by $p_i^1(k) = (\beta + \frac{1-\beta}{k}) \cdot p_i(1)$. Applying this formula yields a non-increasing run-time and non-decreasing work function for each parallel task. In addition, the run-time function is also convex over the interval $[1, m]$, and so is the corresponding work function in the processing time which is required to apply algorithms JZ12 and CHEN13. The left chart of Figure 4 shows examples of different run-time functions that implement this model for different values of $p(1)$ and β.

[§] http://www.grid5000.fr

DOI: 10.1002/cpe

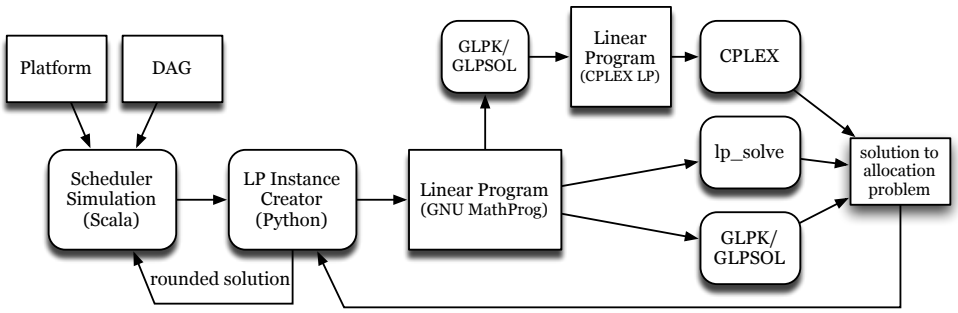


Figure 5. Workflow of LP-based experiments. The total run-time is taken over all stages.

Run-time Model 2: As seen in Figure 1, the run-time of linear algebra functions like PDGEMM is not monotonically decreasing in the number of processors, and it is often larger for an odd number of processors or if the number of processors has no integer square root. We therefore model the second run-time function accordingly, but rely on the general behavior of the run-time function obtained from the Amdahl-based model $p_i^1(k)$. The run-time function of tasks $p_i^2(k)$ following Run-time Model 2 is then defined as follows:

$$p_i^2(k) = \begin{cases} p_i^1(1) & \text{if } k = 1, \\ s_1 \cdot p_i^1(k) & \text{if } k > 1 \wedge k \text{ is odd}, \\ s_2 \cdot p_i^1(k) & \text{if } k > 1 \wedge k \text{ is even, but } \sqrt{k} \text{ not an integer}, \\ p_i^1(k) & \text{otherwise.} \end{cases} \quad (1)$$

s_1 and s_2 are the slowdown factors applied when the number of processors is odd or has no integer square root. In the simulations, we set $s_1 = 1.3$ and $s_2 = 1.1$ to mimic the observed run-time behavior of PDGEMM. The right chart of Figure 4 shows an example how non-monotonic run-time functions can be obtained from Run-time Model 2 using slowdown factors on Run-time Model 1. The shape of these functions changes depending on the original choice of β and the sequential execution time $p^1(1)$.

5.2. *Implementing Linear Programs of Approximation Algorithms*

One main goal of this work was to compare schedules produced by heuristics like CPA, MCPA, or CPA13 to schedules computed by approximation algorithms like JZ06, JZ12, or CHEN13. We found that correctly implementing the LP-based approximation algorithms is a major obstacle to achieve our goal. Most often, the programming languages used to define the linear programs (e.g., GNU MathProg) are not as expressive as the mathematical notation used in the formulas of the original papers.

As this paper should also address practical aspects of conducting comparisons of scheduling algorithms, we sketch the execution workflow used to implement the approximation algorithms in Figure 5. The main software component is the scheduling simulator, which is responsible for (1) reading and parsing the platform and DAG files, (2) executing a selected allocation strategy, (3) running a specific mapping algorithm, and (4) reporting statistics and creating visualizations of the final schedules. This simulator is written in Scala, mainly due to language features and portability reasons.

If the experimenter selects one of the approximation algorithms as strategy to solve the allotment problem, the simulator will call a sub-program "LP Instance Creator" to retrieve the solution. This sub-program to solve the allotment problem is written in Python, as Python makes it relatively easy to generate text files (in our case, text files containing LPs). The Python sub-program receives the details about DAG and platform and uses that knowledge to generate a linear program in the MathProg language, which can be used as input to CPLEX, GLPSOL, or lp_solve. Such intermediate step of generating an additional MathProg program has advantages and disadvantages.

DOI: 10.1002/cpe

Most critical in our evaluation is correctness, and thus having the actual linear program with all constraints in a MathProg file is crucial for debugging. Generating the file containing the LP also improves portability as we can use the same LP as input to different solvers instead of using a specific API provided by each solver. The disadvantage, however, is a possibly longer execution time to build the schedule since the LP needs to be written to disk before the selected LP solver is called.

The MathProg LP can be directly used as input to lp_solve and GLPSOL, but not to CPLEX. Fortunately, GLPSOL can also be used to convert LPs between different LP languages. Thus, we use GLPSOL to convert our generated MathProg LP into a CPLEX LP. This approach adds another level of indirection required for correctness reasons, as we wanted to maintain a single LP generator that produces MathProg code only.

After the MathProg code has been written to disk, our Python program that solves the allotment problem calls the specified LP solver (e.g., CPLEX, GLPSOL) and reads the output files of these solvers. The Python allotment program parses the values of the solution to the linear relaxation of the allotment problem and rounds the solution according to the strategy given by Jansen and Zhang or Chen and Chu. The rounded solution that represents a feasible allotment is returned to the Scala simulator, which can now continue with the mapping step.

5.3. Adjusting Approximation Algorithms

Compared to the LPs given by Jansen and Zhang in [5], we changed one constraint and added another as highlighted in bold in Equation (2). First, the index problem in the sixth constraint ($i = 1, \ldots, m-1$) has been corrected and second, but more importantly, we added the constraint $x_j \leq C_j$, which had already been part of the LP in Zhang's PhD thesis [21, p. 81]. Here, the variables $\Gamma^+(j)$ and $\Gamma^-(j)$ denote the set of successors and predecessors of task j, respectively. Without including this constraint the execution time of input tasks becomes large at times, which in turn would lead to unnecessarily long schedules.

LP of Jansen and Zhang

$$
\begin{array}{rll}
\min & C & \\
\text{such that} & 0 \leq C_j \leq L & \text{for all } j \\
& C_j + x_k \leq C_k & \text{for all } j \text{ and } k \in \Gamma^+(j) \\
& \boldsymbol{x_j \leq C_j} & \textbf{for all } \boldsymbol{j} \text{ s.t. } \boldsymbol{\Gamma^-(j) = \emptyset} \\
& x_j \leq p_j(1) & \text{for all } j \\
& x_{j_i} \leq x_j & \text{for all } j \text{ and } i = 1, \ldots, m \\
& 0 \leq x_{j_i} \leq p_j(i) & \textbf{for all } \boldsymbol{j} \textbf{ and } \boldsymbol{i = 1, \ldots, m-1} \\
& x_{j_m} = p_j(m) & \text{for all } j \\
& \hat{w}_j(x_j) = \sum_{i=1}^{m} \bar{w}_{j_i}(x_{j_i}) & \text{for all } j \\
& P = \sum_{j=1}^{n} p_j(1) & \\
& \sum_{j=1}^{n} \hat{w}_j(x_j) + P \leq W & \\
& L \leq C & \\
& W/m \leq C & \\
& \bar{w}_j(x_{j_m}) = 0 & \text{for all } j \\
& \bar{w}_{j_i}(x_{j_i}) = \left[W_j(i+1) - W_j(i)\right] \dfrac{p_j(i) - x_{j_i}}{p_j(i)} & \text{for all } j \text{ and } i = 1, \ldots, m-1
\end{array}
\tag{2}
$$

DOI: 10.1002/cpe

LP of Chen and Chu As the LP of Chen and Chu proposed in [12] is mainly based on the LP of Jansen and Zhang, we found that the following constraint is also missing in their linear program:

$$x_j \leq C_j \qquad \text{for all } j \text{ s.t. } \Gamma^-(j) = \emptyset \tag{3}$$

We note that—besides adding this constraint to our version of the LP—we implemented Chen and Chu's algorithm for the general assumption (i.e., the processing time is decreasing but not strictly decreasing), and therefore we used $\rho = 0$ and $\mu = \sqrt{2m^2 - 2m}/2$ as rounding parameters.

5.4. Simulation Results

We have conducted three different simulation studies, which are part of the overall evaluation. First, we compare the schedules produced by CPA13 and its competitors for the non-increasing and convex run-time model. Second, we address the scalability of these algorithms by increasing the number of nodes in the DAGs as well as increasing the number of processors per platform. The third set of simulations covers the makespan quality of several scheduling algorithms for the case of arbitrary run-time functions.

5.4.1. Results with Run-time Model 1 The first set of simulations compares scheduling algorithms, which were designed for non-increasing run-time functions, to CPA13. This experiment should answer two questions: (1) What is the overall schedule quality of CPA13 compared with the lower bound? (2) How good are CPA13's solutions compared with solutions obtained from approximation algorithms?

In previous studies that evaluated CPA and some other heuristics, the absolute makespan produced by the different algorithms was used as the metric to compare the schedule quality. The problem is that such comparison lacks a meaningful baseline. In case that two heuristics produce awfully long schedules, one heuristic would still beat the other. Nonetheless, both heuristics would be meaningless in practice. The best solution would be to compare the schedules produced by competing algorithms to the optimal schedule for each case. However, the scheduling problem is strongly NP-hard, and therefore we cannot compute the optimal schedule in reasonable time. We solve this problem by using the lower bound of the scheduling problem as an approximation of the optimum as done, for example, by Albers and Schröder [22]. For the problem of (offline) scheduling moldable tasks with precedence constraints, the length of the critical path and the average work per processor are lower bounds of the makespan. Then, the lower bound of a schedule's makespan is $LB = \max\left\{\sum_{j=1}^{n} w_j(1)/m, L^*\right\}$, where L^* denotes the shortest possible critical path. In order to compute L^* we allocate k_j^* processors to each task v_j with $k_j^* = \arg\min_l p_j(l)$, compute the critical path using this processor allotment and determine its length.

For obtaining the experimental results, the relative run-time improvement in CPA13 was set to $G = 0.01$, i.e., an allocation needs to reduce the task's run-time by at least 1% to be considered. Figure 6 compares the distribution of performance ratios (makespan of schedule / lower bound) of the algorithms under Run-time Model 1. We use regular box-and-whisker plots to visualize the distributions of performance ratios. In these plots, the bottom and the top of a box represent the lower and the upper quartile of the sample data, whereas the line inside the box denotes the median. The whiskers at both ends of the box extend to the largest datum, which is within a distance of $1.5 \cdot$ IQR (interquartile range) from either the lower or the upper quartile. As a consequence, all values that are outside this range are treated as outliers and marked as dots.

We can see that CPA13 achieves the lowest performance ratio for the majority of experiments, but MCPA obtains a slightly better ratio for Strassen DAGs on 48 processors and for layered DAGs on 120 processors. The reason is that CPA13 optimizes not only the makespan but also tries to keep the total work small. For a chain of parallel tasks, MCPA might allocate all processors to one task, whereas CPA13 stops if the efficiency threshold is exceeded. However, as we employ Amdahl's model for the parallel run-time of tasks, the execution time of tasks is strictly decreasing in the number of processors. Thus, MCPA may assign very large allocations with only tiny run-time improvement, but which overall also leads to slightly shorter schedules. The box plot also shows

DOI: 10.1002/cpe

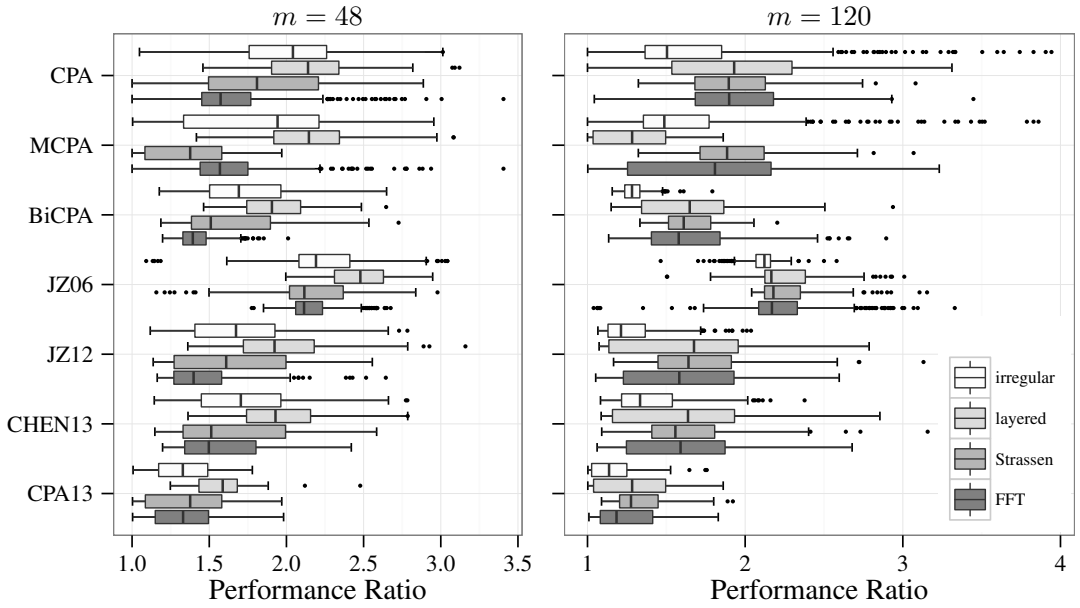


Figure 6. Performance ratios of scheduling algorithms for each DAG class (Run-time Model 1); $m = 48$ (left) and $m = 120$ (right) processors.

that, for each set of DAGs, the performance ratio of CPA13 is improving (decreasing ratio) on the larger machine with $m = 120$ processors. Overall we can conclude that in the cases considered, CPA13 is comparable and mostly better than JZ12, which has a guaranteed approximation ratio of ≈ 3.29. When looking at the performance ratio of the approximation algorithms, we can state that in our simulation study the algorithms JZ12 and CHEN13 produced mostly shorter schedules compared to JZ06. However, there is no clear winner between JZ12 and CHEN13 since the medians of the performance ratios of both are very close to each other.

5.4.2. Scalability Experiments The makespan alone is only one criterion when testing the practical applicability of a scheduling algorithm. Therefore, we also investigate the scalability of the algorithms, i.e., how much time is needed to create a schedule for a given number of processors or nodes (in a DAG). Such study is important to understand the consequence for practical implementations of scheduling algorithms, i.e., we may ask, how long does it take to solve a linear program with $O(mn)$ variables and $O(n^2 + mn)$ constraints? Of course, our practical assessment can only give us a snapshot of the algorithmic performance, as the execution time is strongly dependent on the evolution of the LP solver and the processor generation.

The run-time measurements in this experiments follow the workflow described in Figure 5. Algorithms that do not require a solution of a linear program are entirely implemented in Scala. For determining the time to solution, we measured the time of the entire simulation run, i.e., from starting to read the input files until the final schedule is produced.

The experiments have been conducted on a dedicated machine powered by an Intel(R) Core(TM) i7-3770 CPU running at 3.40 GHz, which has 4 cores and 8 hardware threads. The software parameters were as follows: Linux kernel 3.12-1, Python 2.7, Scala 2.10.3, and OpenJDK 1.7.0_21. As there are many free and proprietary LP solvers available, we have tested CPLEX, GLPK, and lp_solve, but CPLEX (Studio 12.5.1 Linux x86-64, academic) was significantly faster than the free alternatives, and hence, we have used CPLEX exclusively in the experiments presented.

For the scalability experiments we generated a new set of DAGs with the following fixed parameters (cf. Section 5.1): *width* $= 0.8$, *density* $= 0.5$, *regularity* $= 0.8$, and *jump* $= 0$. We only varied the number of nodes per DAG for $n \in \{10, 50, 250, 500\}$ and created 10 samples for each n. In order to obtain large platforms, we simply replicated processors in the machine model with $m = 120$

DOI: 10.1002/cpe

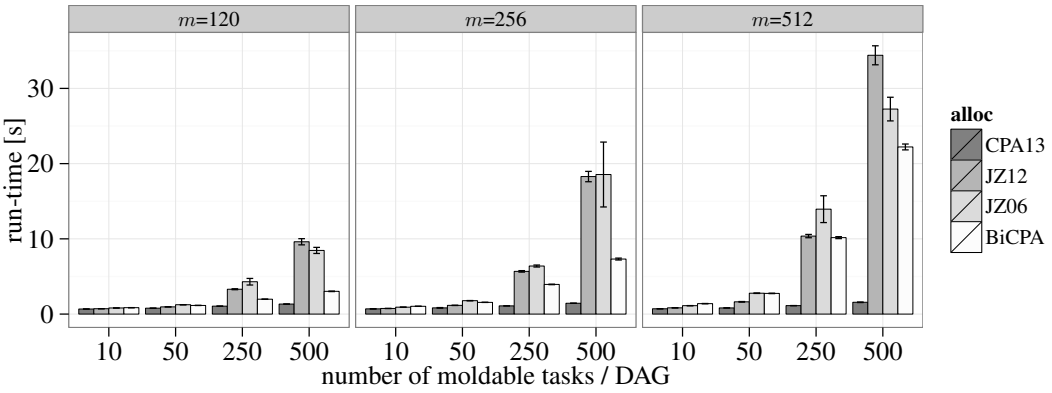


Figure 7. Mean run-time and 95% confidence intervals required to compute schedule for an increasing number of tasks/DAG and number of processors (m).

processors that was used in the previous experiments. Hence, we created two artificial platforms consisting of $m = 256$ and $m = 512$ processors, where each processor has a computational speed of 3.1 GFLOPS.

Figure 7 presents the result of the scalability analysis, in which the mean execution time over 10 repetitions per DAG is reported. Since we have 10 DAGs per number of nodes, each bar represents a summary over 100 runs. We are aware that the sample size (10 different DAGs/number of nodes) is relatively small for a rigorous statistical analysis. However, our goal was to present a first study showing general trends.

The results presented in Figure 7 clearly demonstrate that solving linear programs requires significantly more run-time than CPA13. We chose a linear scale for the y-axis to emphasize the fast growth of the run-time of the LP-based solvers and BiCPA. The charts also show that BiCPA provides weaker scalability compared with CPA13 as it needs to build m different mappings to obtain the final schedule, which is very time-consuming. We also wanted to test whether our approach of first generating and writing a text file (the MathProg LP) to disk has much influence on the overall execution time of the approximation algorithms. From our experiments we can state that this is not the case, as the largest LPs ($n = 500$ and $m = 512$) are about 50 MByte in size, for which our Python program needs roughly 0.1 s to generate and write the file to disk. Thus, this overhead is negligible considering a total execution time of approximately 30 s. We also evaluated the scalability of CHEN13, but for fairness reasons we did not include the results in this article as *our implementation* of CHEN13 was an order of magnitude slower than JZ12 or JZ06. We assume that the run-time deficit of our implementation of CHEN13 is mainly caused by the larger number of variables and sets required to store the run-time and work the virtual tasks in the LP. This should be clarified in future work.

We also note that we changed the *optimizer* in the CPLEX LP for CHEN13 from the generic `optimize` to `primopt`, which forces the use of the simplex optimizer, since we had to deal with unexpected infeasible solutions (negative values in some variables). The IBM CPLEX documentation stated that such error is a result of applying the barrier method.

One could criticize that such a study of the mean execution time is imprecise as several parts could be a target for further optimization. For example, the LP formulation has a significant impact on the execution time of the solver. That is certainly true, but nevertheless, we believe that our experiments give us first insights of how fast such linear programs can be solved in practice for larger instance of scheduling problems (500 tasks, 512 processors).

5.4.3. Results with Run-time Model 2 The second study examines parallel tasks with arbitrary run-time functions. Here, we also include the meta-heuristic EMTS that performs an evolutionary schedule optimization [17]. We also add the algorithms CPA and MCPA in this simulation study. However, both algorithms, CPA and MCPA, require non-increasing run-time functions. The problem

DOI: 10.1002/cpe

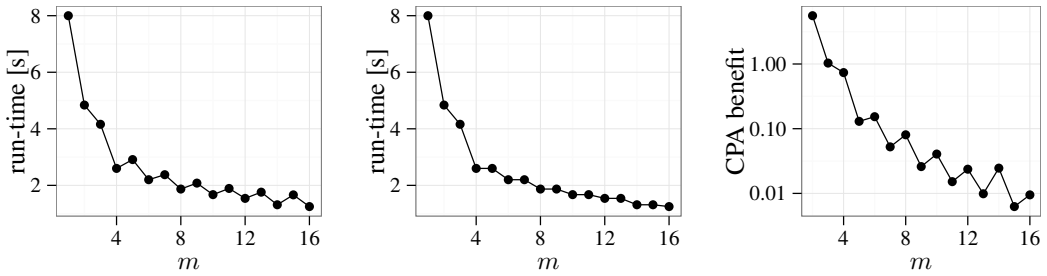


Figure 8. NM+R strategy employed for CPA and MCPA to overcome the problem of non-monotonous run-time functions. The original function (left) is transformed into a monotonously decreasing function (middle). The resulting benefit function is shown on the right-hand side.

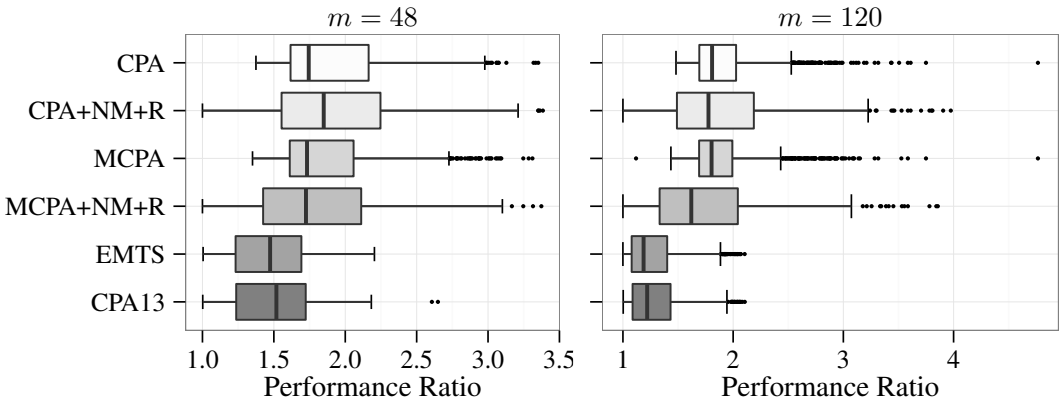


Figure 9. Performance ratios of evaluated scheduling algorithms for all types of DAGs; $m = 48$ (left) and $m = 120$ (right) processors (Run-time Model 2).

of CPA and MCPA with run-time functions that have increasing segments is the computation of the benefit $\left(\frac{p_j(i)}{i} - \frac{p_j(i+1)}{(i+1)}\right)$, which is used to select the best task on the critical path. In case of an increasing run-time function this benefit could become negative, which would lead to unpredictable results. Therefore, we need to make both algorithms non-monotony-aware, which we illustrated in Figure 8. First, we change the run-time function of a parallel task as follows: we use the run-time of the next smaller processor allocation if the run-time increases when the number of processors increases, e.g., if in the original run-time model $p_j(k) < p_j(k+1)$, we set the new run-time $\tilde{p}_j(k+1) = p_j(k)$ in the modified model. Then the following holds: $\forall k, k',\ 1 \leq k,\ k' \leq m,\ k < k' : \tilde{p}(k') \leq \tilde{p}(k)$. However, this newly constructed run-time function $\tilde{p}$ is neither convex in the number of processors nor is the related work function non-decreasing (see middle of Figure 8). For this reason, we cannot apply JZ06 or JZ12 but CPA and MCPA using the run-time function $\tilde{p(j)}$. If we apply the modified model with the monotonically decreasing run-time function, the value of the benefit, computed in CPA and MCPA, will always be positive (see the rightmost chart in Figure 8). Ideally, this benefit function would also be monotonically decreasing and convex, but such modifications are outside the scope of the present paper. We distinguish the versions of CPA and MCPA that apply the modified run-time model from the original versions by appending "NM+R" to the name, where "R" stands for allotment "reduction" in the following case: after the allocation procedure of CPA+NM+R or MCPA+NM+R has finished, processor allotments may be reducible, i.e., a task v_j is allotted to k processors, but there could be a k', $k' < k$ s.t. $\tilde{p}(k') = \tilde{p}(k)$. If there is such a k', we assign k' processors to task v_j since the smaller allotment is not increasing the task's run-time.

Figure 9 shows the distribution of performance ratios over all DAG classes. The box plot reveals

DOI: 10.1002/cpe

that CPA13 produces mostly schedules that are close to the lower bound with a performance ratio of less than 3. EMTS is a meta-heuristic that takes allotments produced by MCPA, CPA, and CPA13 as input and attempts to optimize them. In the simulations, we instantiated an (10 + 100)-EA for EMTS, i.e., $\mu = 10$ parents and $\lambda = 100$ offspring per generation. We stopped EMTS after evaluating 10 generations. It is therefore not surprising that EMTS has a slightly better performance ratio than CPA13. However, we can say that CPA13 already produces very short schedules since EMTS can hardly optimize them further.

As mentioned in the scalability analysis, there is room for improvement in evaluating scheduling algorithms for non-monotonic run-time functions of tasks. It will certainly be interesting to experiment with other types (shapes) of non-monotonic run-time functions for moldable tasks. Nevertheless, our study should serve as a starting point to gain first insight into the problem.

6. DISCUSSION AND CONCLUSIONS

The performance of parallel applications on current hardware depends on many factors such as exploiting deep memory hierarchies. As a result, run-time functions of parallel programs are neither non-increasing nor strictly convex in the number of processors assigned. Hence, we designed a scheduling algorithm for a set of moldable tasks with precedence constraints, which can cope with arbitrary run-time functions of moldable tasks. We identified three key ingredients for producing short schedules for moldable tasks through careful investigation of different problem instances, which are: (1) force task parallelism, (2) avoid allotments with small parallel efficiency and (3) adjust allotments to reduce idle times in the mapping phase. These ingredients have been discovered empirically, but for the sake of readability we did not include a more detailed analysis of each algorithmic factor. Such study would certainly complement the presented paper and could be part of future work.

We showed in a detailed simulation study that the algorithm CPA13 improves schedules not only in the case of arbitrary run-time functions but also for non-increasing run-time functions. One major contribution of the presented work lies in the first comparison of CPA-like heuristics to known approximation algorithms. Our results revealed that CPA13 generates the shortest schedules among the competitors in most cases. Moreover, CPA13 also needs significantly less time to generate the schedules compared to the approximation algorithms, which are based on linear programming. Yet, our results are limited to the cases studied here since CPA13 has no performance guarantee. Finding an approximation ratio for CPA-like algorithm could be addressed in future work.

ACKNOWLEDGEMENTS

We wish to thank the anonymous reviewers of the 2013 Workshop on Scheduling for Parallel Computing (SPC) and the participants of Dagstuhl Seminar 13381 for helpful comments and fruitful discussions on the scheduling problem. In particular, we would like to thank Maciej Drozdowski (Poznan University of Technology) for sharing his knowledge of parallel task scheduling with us. This work was supported by the Austrian Science Fund (FWF): P26124.

REFERENCES

1. Feitelson D, Rudolph L, Schwiegelshohn U, Sevcik K, Wong P. Theory and Practice in Parallel Job Scheduling. *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, *LNCS*, vol. 1291, Springer, 1997; 1–34.
2. Drozdowski M. *Scheduling for Parallel Processing*. Springer, 2009.
3. Tomov S, Nath R, Ltaief H, Dongarra J. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. *Proceedings of the 15th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, 2010; 1–8.
4. Lepère R, Trystram D, Woeginger G. Approximation Algorithms For Scheduling Malleable Tasks Under Precedence Constraints. *International Journal of Foundations of Computer Science* 2002; **13**(04):613–627.
5. Jansen K, Zhang H. An Approximation Algorithm for Scheduling Malleable Tasks under General Precedence Constraints. *ACM Transactions on Algorithms* 2006; **2**(3):416–434.

DOI: 10.1002/cpe

6. Jansen K, Zhang H. Scheduling malleable tasks with precedence constraints. *Journal of Computer and System Sciences* 2012; **78**(1):245–259.
7. Megow N, Günther E, König FG. Scheduling and Packing Malleable and Parallel Tasks with Precedence Constraints of Bounded Width. *Journal of Combinatorial Optimization* 2012; **27**:164–181.
8. van de Geijn RA, Watts J. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency - Practice and Experience* 1997; **9**(4):255–274.
9. Graham RL, Lawler EL, Lenstra JK, Rinnooy Kan A. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics. v5* 1979; **5**:287–326.
10. Leung JYT ((ed.)). *Handbook of Scheduling: Algorithms, Models and Performance Analysis*. Chapman & Hall/CRC, 2004.
11. Skutella M. Approximation Algorithms for the Discrete Time-Cost Tradeoff Problem. *Mathematics of Operations Research* 1998; **23**(4):909–929.
12. Chen CY, Chu CP. A 3.42-approximation algorithm for scheduling malleable tasks under precedence constraints. *IEEE Transactions on Parallel Distributed Systems* 2013; **24**(8):1479–1488.
13. Radulescu A, van Gemund A. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. *Proceedings of the 2001 International Conference on Parallel Processing (ICPP '01)*, 2001; 69–76.
14. Bansal S, Kumar P, Singh K. An improved two-step algorithm for task and data parallel scheduling in distributed memory machines. *Parallel Computing* 2006; **32**(10):759–774.
15. Hunold S. Low-Cost Tuning of Two-Step Algorithms for Scheduling Mixed-Parallel Applications onto Homogeneous Clusters. *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2010; 253 – 262.
16. Desprez F, Suter F. A Bi-criteria Algorithm for Scheduling Parallel Task Graphs on Clusters. *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2010; 243–252.
17. Hunold S, Lepping J. Evolutionary Scheduling of Parallel Tasks Graphs onto Homogeneous Clusters. *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER)*, 2011; 344–352.
18. Roderus M, Berariu A, Bungartz HJ, Krüger S, Matveev AV, Rösch N. Scheduling parallel eigenvalue computations in a quantum chemistry code. *Proceedings of the 16th International Euro-Par Conference*, *Lecture Notes in Computer Science*, vol. 6272, D'Ambra P, Guarracino MR, Talia D (eds.), Springer, 2010; 113–124.
19. Nagarajan V, Wolf JL, Balmin A, Hildrum K. Flowflex: Malleable scheduling for flows of MapReduce jobs. *Proceedings of the ACM/IFIP/USENIX 14th International Middleware Conference*, *Lecture Notes in Computer Science*, vol. 8275, Eyers DM, Schwan K (eds.), Springer, 2013; 103–122.
20. Suter F. DAGGEN: A synthetic task graph generator. https://github.com/frs69wq/daggen.
21. Zhang H. Approximation algorithms for min-max resource sharing and malleable tasks scheduling. PhD Thesis, University of Kiel 2004.
22. Albers S, Schröder B. An Experimental Study of Online Scheduling Algorithms. *Journal of Experimental Algorithmics* 2002; **7**:3.

DOI: 10.1002/cpe