Scheduling Moldable Tasks with Precedence Constraints and Arbitrary Speedup Functions on Multiprocessors

Sascha Hunold(✉)

Research Group Parallel Computing,
Vienna University of Technology, Vienna, Austria
hunold@par.tuwien.ac.at

Abstract. Due to the increasing number of cores of current parallel machines, the question arises to which cores parallel tasks should be mapped. Thus, parallel task scheduling is now more relevant than ever, especially under the moldable task model, in which tasks are allocated a fixed number of processors before execution. Scheduling algorithms commonly assume 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.

Keywords: Multiprocessor scheduling · Homogeneous processors · Moldable tasks · Makespan optimization · Speedup functions

1 Introduction

The problem of scheduling parallel tasks has been intensively studied, and it originally stems from the need of improving the utilization of massively parallel processors (MPPs) [1]. Researchers attempted to understand the implications of different job scheduling strategies on the utilization of parallel systems theoretically and practically. 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.

We focus on the *moldable* task model. The reason is mostly practical, since the malleable model would require additional effort from programmers, e.g., to

R. Wyrzykowski et al. (Eds.): PPAM 2013, Part II, LNCS 8385, pp. 13–25, 2014.
DOI: 10.1007/978-3-642-55195-6_2,

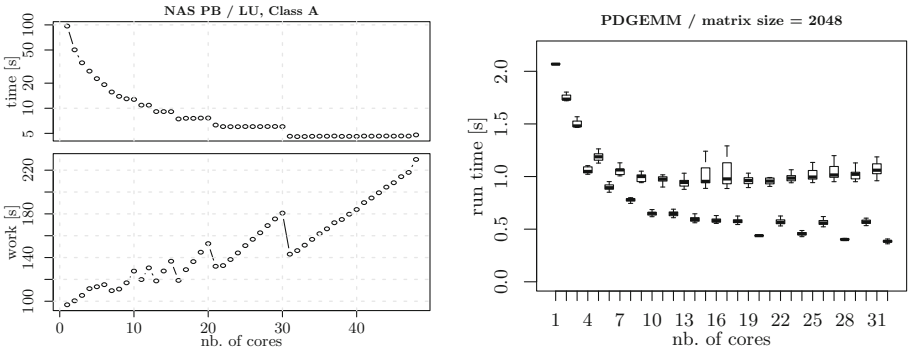

Fig. 1. Left: work (top) and run-time (bottom) of LU benchmark (4 sockets, 48 cores, AMD Opteron 6168); Right: execution times of PDGEMM (20 runs per core count, GBit Ethernet, AMD Opteron 6134)

redistribute data or synchronize thread groups. Pthreads or OpenMP programs are typical examples of moldable tasks as users can specify the number of threads before the execution of a parallel program. MPI applications are examples of moldable programs for distributed memory machines.

Today, researchers in parallel computing face the question of how to program the available processors (or cores) efficiently. One 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].

To execute moldable tasks of a DAG, a scheduling algorithm has to determine (1) the next task to be executed and (2) the set of processors to which a task is mapped. In the scheduling literature, this is known as *scheduling problem for moldable tasks with precedence constraints* [2] (sometimes also called malleable task scheduling [4–7]). 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. 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., at 21 or 31 cores. The chart on the right shows the run-time of PDGEMM from ScaLAPACK (using GotoBLAS). 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]. One could solve this problem of the 0run-time function by using $(k-1)$ instead of k processors if the run-time on $(k-1)$ processors is smaller. The resulting run-time function would be non-increasing, but the work function could still be decreasing (similar to the plot on the left-hand side).

Therefore, we propose an *algorithm for scheduling non-preemptive moldable tasks with precedence 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., we investigate $P|any, NdSub, prec|C_{\max}$ and $P|any, prec|C_{\max}$, where P denotes the number of processors, *any* the moldable tasks, *prec* the precedence constraints, and $NdSub$ the nondecreasing sublinear speedup[1]. Our objective is to minimize the makespan $C_{\max}$.

We make several contributions to moldable task scheduling. First, we propose a *new algorithm* (CPA13)[2] that *supports arbitrary run-time functions* of moldable tasks. We show through simulation that our algorithm is competitive in the 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 when *the run-time function of each parallel task is non-increasing*. Previous studies of moldable task scheduling algorithms use the absolute makespan to compare heuristics. However, such analysis provides little evidence of the schedule quality. Hence, as third contribution, we compare algorithms by their *performance ratio*, which is the *ratio of makespan and lower bound*.

Section 2 introduces notation and Sect. 3 discusses related work. We introduce the new scheduling algorithm and show complexity results in Sect. 4, while Sect. 5 presents the simulation results before we conclude in Sect. 6.

2 Definitions and Notation

We consider the problem of scheduling n moldable tasks with precedence constraints on m identical processors under 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 scheduler. The application is represented as 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 $i \leq k$. 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 $W = \sum_{v_i} p_i(\alpha_i)\alpha_i$.

[1] For more details on notation see [2,9,10].

[2] Critical Path and Area-based Scheduling (CPA), "13" refers to the present year.

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 exist. 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 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]. The algorithms presented in [4–6] require monotony of runtime and work, and the most recent algorithm (with approximation ratio ≈ 3.29) also 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].

Radulescu and van Gemund used similar observations as reported in [4,5] for solving the allotment problem. Thus, the 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} [12]. CPA starts by allocating a single processor to each task, inspects the tasks on the critical path, and then adds one processor to the task that decreases the average processor utilization (runtime / number of processors) the most. 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 [13]. More precisely, the allocation routine of CPA often produces large processor allotments, with the consequence that tasks—which can potentially be 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, no more processors are allotted to a task if already m processors have been allotted to 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 significantly [14].

Desprez and Suter attempted to optimize both the makespan and the total work when scheduling a DAG [15]. They proposed 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, yet it increases 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. Günther et al. presented an FPTAS for this scheduling problem [7]. As the general problem is strongly NP-hard, they looked at DAGs with bounded width and developed a dynamic program. For practical applicability, the FPTAS has a rather large complexity of $O((\frac{n^3}{\epsilon})^{2\omega} m^{2\omega})$, where

Table 1. Overview of notation used in the present article

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
$p_j(k)$	–	run-time of task j with k processors	b_j	–	benefit of task j
r_j	–	rel. run-time improvement	l_j^b	–	bottom level of task v_j
l_j^p	–	precedence level of task v_j	$\tilde{m}_d$	–	nb. processors in prec. level d

ω denotes the maximum width of a DAG. In previous work, we already addressed the challenge of arbitrary run-time functions by using an evolutionary algorithm (EA) to find short schedules [16]. The proposed algorithm EMTS takes allotment solutions of several heuristics, like CPA and MCPA, and optimizes them using an $(\mu+\lambda)$-EA. When generating and evaluating many offspring, EMTS can find short schedules, while having the disadvantage of an increased run-time.

In sum, efficient heuristics and approximation algorithms only exist for non-increasing run-time and non-decreasing work functions, and previous algorithms without such limitations have high computational demands.

4 Scheduling Algorithm

Our proposed scheduling algorithm applies concepts of the algorithms discussed before, e.g., reducing the critical path while keeping the overall work small. Lepère et al. and Jansen/Zhang also applied this concept in their algorithms. We define the following requirements for our scheduling algorithm: several published articles addressed the problem of CPA that allotments can grow too big regardless of their speedup. To solve this problem, we introduce the *relative run-time threshold* G, which defines the minimum runtime improvement that a larger allotment needs to provide to be considered as possible solution. As shown later, this threshold is key for short and compact schedules. Task parallelism should be conserved as much as possible. To do so, MCPA checks all the visited nodes in a certain DAG depth, but may unmark once visited nodes. In contrast, our algorithm considers all allotments of those nodes that were once marked. In addition, the mapping function that selects processors for ready tasks has often been overlooked in previous studies. Since we showed that reducing processor allotments in the mapping step can significantly improve the overall schedule [14], CPA13 applies a binary search strategy to find a possibly smaller task allotment that does not increase the estimated completion time.

4.1 Pseudocode

Algorithm 1 presents the allotment function of our algorithm named CPA13. For an easier comprehension we summarize the notation in Table 1. Let us highlight the main steps of the algorithm. In the initialization phase, each task is allotted

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 for which:
$\forall a_i, a_k \in A_j, i < k : p_j(i) > p_j(k)$
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 (k, b_{j_k}) in list of possible allotments for task v_j
12: **while** $L > W/m$ **do**
13: $V_{CP} \leftarrow$ collect tasks on critical path
14: $(v_b, \tilde{\alpha_b}, b_b) \leftarrow (\mathbf{nil}, m, 0.0)$ // initialize current best temporary values
15: $\tilde{m}_d \leftarrow \sum_{v_l \in \tilde{V}} \alpha_l$ **where** $\tilde{V} = \{v_k \in V \text{ s.t. } l_k^b = d \wedge v_k \text{ marked VISITED } \}$
16: **for all** $v_j \in V_{CP}$ **do**
17: $b_{j_k}, a_{j_k} \leftarrow$ benefit and size of task's v_j next larger allocation
18: $s \leftarrow a_{j_k} - \alpha_j$ // absolute increase in number of processors
19: **if** $\tilde{m}_{d_j} + s \leq m \wedge b_{j_k} > b_b$ **then**
20: $(v_b, \tilde{\alpha_b}, b_b) \leftarrow (v_j, a_{j_k}, b_{j_k})$ // current best
21: **if** $v_b \neq \mathbf{nil}$ **then**
22: $\alpha_b \leftarrow \tilde{\alpha_b}$ // increase allotment of task v_b
23: mark v_b as VISITED
24: recompute L and W
25: **else**
26: **break** // terminate while loop

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).

In the second phase (line 12), we compute the critical path and select the task with the greatest benefit value. We allot more processors to this task unless the number of processes in this precedence level is exceeded (lines 19–20) (The precedence level denotes the shortest path to a node from the source node). After changing the allotment of one task on the critical path, we need to recompute L and W. The allotment process repeats until either the critical path L is smaller than (or equal to) W/m or no more task satisfies the precedence level constraint or provides additional run-time benefit.

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, this allotment of v_j might be packed (decreased in size) 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.

Algorithm 2. CPA13 mapping procedure

1: **while not** all tasks scheduled **do**
2: $v_j \leftarrow$ find tasks with maximum l_j^b
3: LET $C_j = \tau_j + p_j(\alpha_j)$ be the completion time of v_j if v_j is allocated α_j processors that become available first at time τ_j
4: $\alpha_i \leftarrow$ use binary search to find an allocation $\alpha_i \leq \alpha_j$ s.t. $\tau_i + p_j(\alpha_i) \leq C_j$
5: schedule v_j onto first α_i processors that become available

Table 2. Summary of complexity results, "t.p." stands for "this paper"

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

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 (($O(n+e)$). The outer loop (line 12) is executed at most $n \cdot m$ times since then each tasks will have m processors allotted to it. Thus, the complexity of the allocation phase is $O(nm(n+e))$.

The mapping procedure (Algorithm 2) first extracts the task with 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 ($O(m \log m)$). Upon mapping a task, the algorithm visits every outgoing edge to detect ready tasks. 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 2. JZ06 and JZ12 denote the algorithms of Jansen and Zhang from 2006 [5] and 2012 [6]. The authors state that the LIST scheduling function requires $O(nm)$ operations, while $LP(p, q)$ denotes "the time to solve a linear program with p variables and q constraints" [5].

As the number of edges may be greater than mn, we updated this run-time to $O(mn+e)$. The suffix "NM+R" behind CPA and MCPA identifies our modified versions, which are aware of possibly increasing run-time functions (discussed in Sect. 5). The evolutionary algorithm EMTS is input-dependent as it takes solutions of other heuristics for obtaining the initial population, and its run-time grows with the number of generations 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

We use simulation to evaluate CPA13 for two reasons: (1) Simulations allows us to obtain a statistically significant number of results. (2) Not many truly moldable applications exist, which would limit the variety of experiments.

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 random DAGs. Strassen's matrix multiplication algorithm and the Fast Fourier Transformation (FFT) are examples of *application-oriented DAGs*. To obtain different computation and scalability ratios, we keep the shape of these DAGs fixed but change the number of operations of each task. The *synthetic DAGs* are generated with DAGGEN [17] and contain 20, 50 or 100 nodes. Four parameters influence the DAG generation process: the *width* controls how many task can run in parallel, the *regularity* defines the uniformity of the number of tasks per DAG level, *density* specifies the number of edges, and *jump* denotes if and how many DAG levels an edge (arc) may span. In total, we created 400 FFT, 100 Strassen, 108 layered and 324 irregular DAGs. *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 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, we apply Amdahl's model and pick the non-parallelizable fraction β (see next section) of $f(d)$, which is selected randomly from a uniform distribution between 0 and 0.25. Due to the page limit we refer to [15,16] for more details.

The platform model has two parameters: (1) the number of processors m and (2) the speed of the processor (in GLFOPS). We use two machine models in the simulations. The first models a Grid'5000 cluster (Grelon) with $m = 120$ processors providing 3.1 GFLOPS each (obtained with HP-LinPACK). The

other machine has $m = 48$ processors running at 6.7 GFLOPS (measured with GotoBLAS2), modeling a shared-memory system at TU Vienna.

We apply two different run-time models to parallel tasks in our simulation.

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 apply Amdahl's law to obtain a run-time model. Let $p_i(1)$ be the sequential run-time of task v_i, determined by the ratio of the number of operations 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 bounded 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 $[1, m]$, which is required to apply algorithm JZ12.

Run-time Model 2: Figure 1 has shown that the run-time of PDGEMM is larger with an odd number of processors or if the number of processors has no integer square root. We model the second runtime function accordingly, but base it on the Amdahl model $p_i^1(k)$. Equation (1) shows the runtime function $p_i^2(k)$, which may increase if allocations get larger.

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

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 reflect the observed run-time behavior of PDGEMM.

5.2 Simulation Results

Run-time Model 1. The first set of simulations compares algorithms that were designed for non-increasing run-time functions with CPA13. This experiment answers 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 of approximation algorithms?

In previous studies of CPA, algorithms have been compared without a baseline, so it was uncertain whether experimental findings have significant impact. Since the problem is strongly NP-hard, we use the lower bound as an approximation of the optimum as done by Albers and Schröder [18]. The length of the critical path and the average work per processor are lower bounds of the makespan. Thus, the lower bound of the makespan is $LB = \max\left\{\sum_{j=1}^{n} w_j(1)/m, L^*\right\}$, where L^* denotes the shortest possible critical path. To compute L^* we allocate k processors to task v_j with $k = \arg\min_l p_j(l)$, then we compute the critical path using this processor allotment and determine its length.

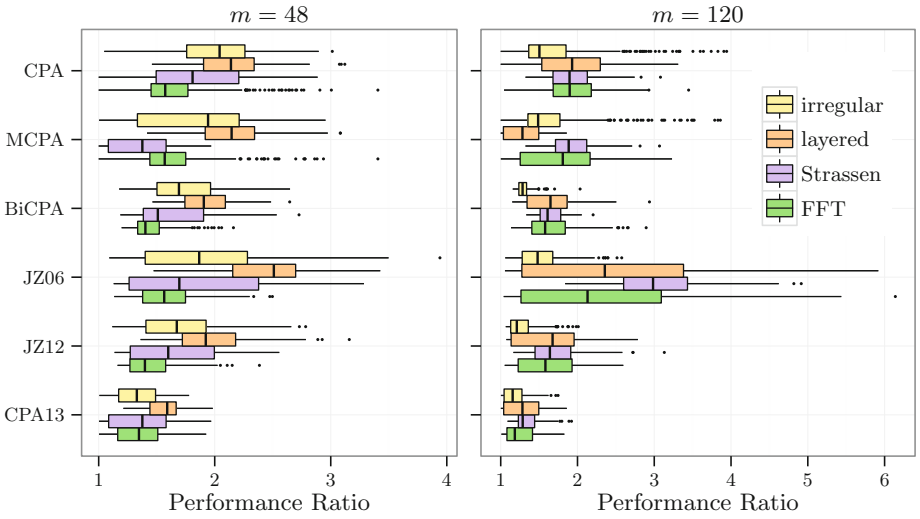

Fig. 2. Performance ratios of scheduling algorithms for each DAG class (Run-time Model 1); $m = 48$ (left) and $m = 120$ (right) processors

Figure 2 compares the performance ratio (makespan of algorithm / lower bound) of the algorithms under Run-time Model 1. The CPA13 threshold for the relative gain was set to $G = 0.01$, i.e., an allocation needs at least 1 % of runtime improvement to be considered. We can see that CPA13 achieves the lowest performance ratio, while 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 may allocate all processors to some task, whereas CPA13 stops if the efficiency threshold is exceeded. Thus, MCPA produces larger allocations with slightly shorter runtime, but which leads to slightly shorter schedules. The graph also shows that the performance ratio of CPA13 decreases on the bigger machine. Overall, in the cases considered, CPA13 is comparable and mostly better than JZ12, which has an approximation ratio of ≈ 3.29. We also experimented with larger DAGs, for which the linear programs of JZ06 and JZ12 have many constraints. On an Intel Core i7 (2.3 GHz) using IBM CPLEX, the time for solving instances with 1000 tasks and 120 processors took on average 49 s with JZ12 and 28 s with JZ06. In contrast, CPA13 produces solution for these instances in an average time of 2.5 s, which shows its scalability.

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 [16].

Since CPA and MCPA only assume non-increasing run-time functions, we make both algorithms non-monotony-aware. To do so, we change the run-time function of a parallel task as follows: We use the run-time of the next smaller

algorithm	m	work ratio	$\sqrt{C_{\max} \cdot W}$ ratio
CPA13	120 / 48	3.88 / 2.01	2.17 / 1.75
JZ12	120 / 48	3.17 / 1.85	2.16 / 1.74
JZ06	120 / 48	5.62 / 2.45	2.94 / 2.02
BiCPA	120 / 48	2.09 / 1.62	1.84 / 1.67
MCPA	120 / 48	4.05 / 2.07	2.57 / 1.99
CPA	120 / 48	4.08 / 2.08	2.67 / 2.07

Fig. 3. Performance ratios of evaluated scheduling algorithms for several DAG classes; $m = 48$ (left) and $m = 120$ (right) processors (Run-time Model 2)

processor allocation if the run-time increases when the number of processors increases, e.g., if in the original execution time model p_j the execution time of $p_j(k) < p_j(k+1)$, we set in the modified model $\tilde{p}_j(k+1) = p_j(k)$. Then the following holds: let $1 \leq k, k' \leq m, k < k'$, so $\tilde{p}(k') \leq \tilde{p}(k)$. Yet, this newly constructed run-time function $\tilde{p}$ is neither convex in run-time nor non-decreasing in work. For this reason, we cannot apply JZ06 or JZ12 but we can use CPA and MCPA with $\tilde{p}$. We distinguish them 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 has k processors allotted, but $\exists k'$, $k' < k$ for which $\tilde{p}(k') = \tilde{p}(k)$. If so, we assign k' processors to task v_j since the smaller allotment is not increasing the task's run-time.

Figure 3 shows the distribution of performance ratios over all DAG classes. The chart reveals that CPA13 produces mostly schedules that are close to the lower bound with a performance ratio of less than three. EMTS is a metaheuristic 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 EA generations. It is therefore not surprising that EMTS has a slightly better performance ratio than CPA13. However, we can state that CPA13 already produces very short schedules since EMTS hardly can optimize them further.

6 Discussion and Conclusions

The performance of parallel applications on current hardware depends on many factors such as deep memory hierarchies. As a result, run-time functions of a parallel program depending in the number of processors are neither non-increasing or strictly convex. Hence, we designed a scheduling algorithm for moldable tasks with precedence constraints and for arbitrary run-time functions. We identified 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.

We showed in a detailed simulation study that the algorithm CPA13 improves schedule not only in the case of arbitrary run-time functions but also for non-increasing run-time functions. One major contribution lies in the comparison of CPA and its variants to known approximation algorithms. Our results revealed that CPA13 generates the shortest schedules among the competitors in most cases. Yet, our results are limited to the cases studied here since CPA13 has no performance guarantee, which could be addressed in future work.

References

1. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and practice in parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291, pp. 1–34. Springer, Heidelberg (1997)
2. Drozdowski, M.: Scheduling for Parallel Processing. Springer, London (2009)
3. Tomov, S., Nath, R., Ltaief, H., Dongarra, J.: Dense linear algebra solvers for multicore with GPU accelerators. In: HIPS Workshop, pp. 1–8 (2010)
4. Lepère, R., Trystram, D., Woeginger, G.: Approximation algorithms for scheduling malleable tasks under predence constraints. Int. J. Found. Comput. Sci. **13**(04), 613–627 (2002)
5. Jansen, K., Zhang, H.: An approximation algorithm for scheduling malleable tasks under general precedence constraints. ACM Trans. Algorithms **2**(3), 416–434 (2006)
6. Jansen, K., Zhang, H.: Scheduling malleable tasks with precedence constraints. J. Comput. Syst. Sci. **78**(1), 245–259 (2012)
7. Günther, E., König, F.G., Megow, N.: Scheduling and packing malleable and parallel tasks with precedence constraints of bounded width. J. Comb. Optim. **27**(1), 164–181 (2014)
8. van de Geijn, R.A., Watts, J.: SUMMA: scalable universal matrix multiplication algorithm. Concurr. Pract. Exp. **9**(4), 255–274 (1997)
9. Graham, R.L., Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Ann. Discrete Math. **5**, 287–326 (1979)
10. Leung, Y.J.T. (ed.): Handbook of Scheduling: Algorithms, Models and Performance Analysis. Chapman & Hall/CRC, Boca Raton, FL, USA (2004)
11. Skutella, M.: Approximation algorithms for the discrete time-cost tradeoff problem. Math. Oper. Res. **23**(4), 909–929 (1998)
12. Radulescu, A., van Gemund, A.: A low-cost approach towards mixed task and data parallel scheduling. In: ICPP, pp .69–76 (2001)
13. Bansal, S., Kumar, P., Singh, K.: An improved two-step algorithm for task and data parallel scheduling in distributed memory machines. Parallel Comput. **32**(10), 759–774 (2006)
14. Hunold, S.: Low-cost tuning of two-step algorithms for scheduling mixed-parallel applications onto homogeneous clusters. In: CCGrid, pp. 253–262 (2010)
15. Desprez, F., Suter, F.: A bi-criteria algorithm for scheduling parallel task graphs on clusters. In: CCGrid, pp. 243–252 (2010)

16. Hunold, S., Lepping, J.: Evolutionary scheduling of parallel tasks graphs onto homogeneous clusters. In: CLUSTER, pp. 344–352 (2011)
17. Suter, F.: DAGGEN: a synthetic task graph generator. https://github.com/frs69wq/daggen
18. Albers, S., Schröder, B.: An experimental study of online scheduling algorithms. J. Exp. Algorithmics **7**, 3 (2002)