J. Parallel Distrib. Comput. 72 (2012) 693–704

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

An effective approximation algorithm for the Malleable Parallel Task Scheduling problem☆

Liya Fan [a,b,c,*], Fa Zhang [a], Gongming Wang [a,b], Zhiyong Liu [a]

[a] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China
[b] Graduate University of Chinese Academy of Sciences, Beijing, 100049, China
[c] IBM China Research Laboratory, Beijing, 100193, China

ARTICLE INFO

Article history:
Received 8 March 2011
Received in revised form
17 January 2012
Accepted 24 January 2012
Available online 2 February 2012

Keywords:
Scheduling algorithm
Approximation algorithm
Parallel computing

ABSTRACT

The Malleable Parallel Task Scheduling problem (MPTS) is an extension of one of the most classic scheduling problems ($P\|C_{max}$). The only difference is that for MPTS, each task can be processed simultaneously by more than one processor. Such flexibility could dramatically reduce the makespan, but greatly increase the difficulty for solving the problem. By carefully analyzing some existing algorithms for MPTS, we find each of them suitable for some specific cases, but none is effective enough for all cases. Based on such observations, we introduce some optimization algorithms and improving techniques for MPTS, with their performance analyzed in theory. Combining these optimization algorithms and improving techniques gives rise to our novel scheduling algorithm OCM (Optimizations Combined for MPTS), a 2-approximation algorithm for MPTS. Extensive simulations on random datasets and SPLASH-2 benchmark reveal that for all cases, schedules produced by OCM have smaller makespans, compared with other existing algorithms.

1. Introduction

Of the entire myriad scheduling problems, those aimed at load balancing have always been the focus of attention, because they are often directly related to the parallel efficiency of many applications. For instance, a frequently encountered problem is to schedule a set of independent tasks to achieve load balancing on homogeneous parallel computers, given the processing time of each task. This is one of the most classic scheduling problems. According to the standard notation for scheduling problems [16], it is denoted by $P\|C_{max}$. The problem discussed in this study, namely the Malleable Parallel Task Scheduling problem (MPTS), is another common scheduling problem for load balancing, and it is an extension of $P \parallel C_{max}$.

For MPTS, a set of n independent tasks $T = \{T_1, T_2 \ldots T_n\}$ are processed by a set of m identical processors $M = \{M_1, M_2 \ldots M_m\}$. The tasks are processed on any processor(s) non-preemptively. In other words, once a task starts to execute, it cannot be interrupted until finished. Each processor may process at most one task at a time. The objective of the problem is to minimize the makespan, i.e. the finishing time of the last completed task. The only difference between MPTS and $P\|C_{max}$ is that for MPTS, each task may be processed simultaneously by more than one processor, so its processing time is a function of the number of processors allotted to it.

If the processing time of T_j on p_j processors is $t_j(p_j)(p_j = 1, 2 \ldots m)$, then the following assumptions naturally hold for most practical cases [4]:

$$t_j(p_j) \geq t_j(p_j + 1) \quad p_j = 1, 2 \ldots m - 1$$
$$p_j \cdot t_j(p_j) \leq (p_j + 1) \cdot t_j(p_j + 1) \quad p_j = 1, 2 \ldots m - 1.$$

The first assumption holds simply because more processors would bring more computing resources, and hence lead to shorter processing time. We call it the *decreasing time assumption*. Generally, more processors would incur more communication and synchronization costs, which would increase the total amount of work for the task. This is the rationale for the second assumption.

Abbreviations: MPTS, the Malleable Parallel Task Scheduling problem; OCM, Optimizations Combined for MPTS; NPTS, the Non-malleable Parallel Task Scheduling problem; LPT, Longest Processing Time (algorithm); OM, Optimizing the Middle (algorithm); OT, Optimizing the Top (algorithm); OB, Optimizing the Bottom (algorithm); ER, Effectiveness Ratio.

☆ This work was supported in part by the National Natural Science Foundation of China (under 61020106002 and 61161160566), and CAS Knowledge Innovation Key Project (KGCX1-YW-13).

* Corresponding author at: Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China.

E-mail addresses: fanliya@cn.ibm.com (L. Fan), zf@ncic.ac.cn (F. Zhang), wanggongming@ict.ac.cn (G. Wang), zyliu@ict.ac.cn (Z. Liu).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.01.011

We call it the *increasing work assumption*. For MPTS, both of these assumptions hold and the processing time of each task on any number of processors is given as the input of the problem. This completes the formal definition of MPTS. It has been proved to be NP-hard in the strong sense [28].

Although there is only one difference between MPTS and $P\|C_{max}$, algorithms and schedules for the two problems differ greatly. For example, the List algorithm guarantees to produce schedules with makespan within twice of the optimum for $P\|C_{max}$ [14], but for MPTS, it only guarantees to produce makespan within m times the optimum (See Section 3.1). In fact, for many cases, schedules produced for MPTS have much smaller makespans compared with $P\|C_{max}$. This is due to the extra flexibility of MPTS that allows tasks to be processed by more than one processor. Because of such advantage of MPTS, it has been adopted by more and more practical applications, especially applications on multi/many-core systems, where processing a task by more than one processor incurs only a little cost. Although MPTS has solutions with smaller makespans, it is more difficult to solve: the polynomial time approximation scheme for $P\|C_{max}$ has been introduced in 1987 [19], while the best theoretical result for MPTS is only a 3/2-approximation algorithm [32].

For the sake of clarity, we give the following notations and definitions. The processing time on one processor is called the task's sequential processing time. The work of a task is defined as its processing time multiplied by the number of processors allotted to it. Given a schedule, a processor is defined as a critical processor if it is finished last in the schedule. Similarly, a task is called a critical task of the schedule if it is finished the latest. Note that there may be more than one critical processor or task in a schedule. An allotment for a task is a positive integer between 1 and m indicating the number of processors allotted to it, and an allotment for a set T of n tasks is an n-dimensional vector $p = (p_1, p_2 \dots p_n)^T$, where p_j is the allotment for T_j. A sequential task is one that is processed by only one processor, and a parallel task is processed by at least two. Let C^*_{max} and $(p^*_1, p^*_2 \dots p^*_m)^T$ denote the makespan and allotment of an optimal schedule, respectively. Similarly, the makespan produced by algorithm A is denoted by C^A_{max}.

In this paper, some existing algorithms for MPTS are discussed, with their features analyzed in detail. Our novel optimization algorithms and improving techniques for MPTS are based on such analyses. Our effective scheduling algorithm OCM (Optimizations Combined for MPTS) is in turn based on these optimization algorithms and improving techniques. By effective, we mean that the algorithm is capable of producing schedules with high qualities (small makespans, balanced loads, etc.). OCM is a 2-approximation algorithm for MPTS. Simulations on both random datasets and the SPLASH-2 benchmark suite reveal that schedules produced by OCM have smaller makespans, compared with other existing algorithms.

The rest of the paper is organized as follows: Section 2 describes the previous work for MPTS. Section 3 discusses some existing algorithms, and analyzes their features. Our novel algorithms and improving techniques are introduced in Section 4. In Section 5, results of extensive simulations are shown. Finally, we summarize and conclude in Section 6.

2. Related work

2.1. Algorithms for special cases of MPTS

A common special case of MPTS is named the Non-malleable Parallel Task Scheduling problem (NPTS). The only difference between MPTS and NPTS is that for NPTS, the allotments for all tasks are given as part of the problem instance. Since many algorithms for MPTS are based on algorithms for NPTS, we briefly describe some algorithms for NPTS. NPTS has been proved to be NP-hard in the strong sense, even if all tasks have unit processing times [5]. The list scheduling algorithm introduced by Garey and Graham can be used for NPTS [13] to obtain a schedule with makespan within twice of the optimum. Amoura, et al. gave a polynomial time approximation scheme for NPTS, which requires the number of processors fixed [2].

In a common special case of NPTS, the indices of processors allotted to each task are required to be contiguous. This is called NPTS on a line [3] or strip-packing [29]. Some heuristics have been introduced for this case, such as Coffman et al.'s heuristics NFDH (Next-Fit Decreasing Height) [10], and FFDH (First-Fit Decreasing Height) [10], and Lodi's heuristic BFDH (Best-Fit Decreasing Height) [29]. All these heuristics assign jobs by levels. The Bottom–Left algorithm introduced by Baker et al. is a 3-approximation algorithm for strip-packing [3]. Sleator's algorithm reduced the approximation ratio to 2.5 [34], and Steinberg's algorithm further reduced the ratio to 2 [35]. In 2000, Kenyon and Remila proposed an asymptotic fully polynomial time approximation scheme for strip-packing [24]. The scheduling algorithm for hierarchical clusters [8] can also be used for this case, with an approximation ratio of 2.5.

Similarly, NPTS can also be defined on a mesh, for which the processors are arranged and connected as a 2D mesh, and processors allotted to each task must be a sub-mesh. In 1994, Feldmann, et al. introduced their 14-approximation algorithm for this case [12].

For another special case of MPTS, the work of a task remains unchanged as the number of processors increases, so the speedup equals the number of allotted processors. Such tasks are called work preserving [28]. The earliest completion time (ECT) heuristic by Wang et al. was introduced for this case [37], and further analyses to this heuristic can be found in [11,38].

In some cases, the amount of processors allotted to a task may be a real number between 0 and m, so the processing time of each task is a continuous function. The problem of minimizing the makespan under this continuous model was studied in [39,40]. Prasanna et al. studied the problem with precedence constraints among tasks [33]. Results from the continuous model are also of significant value to the design of algorithms for MPTS. First, the solution to the continuous model can be adjusted to produce the solution to MPTS. For example, the algorithm of [6] is a 2-approximation algorithm for MPTS, and it constructs the solution based on the solution to the continuous model. Second, some insights gained from the continuous model are helpful to design algorithms for MPTS.

Other algorithms for special cases of MPTS include Chen and Lee's polynomial time approximation scheme, which requires the number of processors fixed [9]. The algorithm by Blazewicz, et al. is suitable for problem instances with no more tasks than processors [7]. Krishnamurti and Ma's algorithm deals with the case for which the allotment of each task is bounded from above [27]. Jansen and Porkolab introduced an algorithm for the special case of NPTS where task execution can be preempted [22]. Kovalyov et al.'s algorithm is also for preemptive tasks, and the allotment can be changed during execution [26]. The algorithm by Jansen and Thöle is for the special case where the number of processors is polynomially bounded by the number of tasks [23]. The algorithm introduced by Havill and Mao schedules online malleable tasks with setup times [18].

2.2. Algorithms for MPTS

In 1990, Banerjee introduced a 2-approximation algorithm [4] for MPTS based on the LPT (longest processing time) algorithm [15]. The algorithm has a low time complexity $O(n \log n + nm \log m)$, and is quite effective, according to our simulations in Section 5. We believe a major reason is that the algorithm sorts

tasks in descending order of the processing time. This technique has been proved to be beneficial [15] for the case where each task requires exactly one processor (i.e. the $P\|C_{max}$ problem). Banerjee's algorithm generally allots small numbers of processors to tasks, so it is suitable for poorly malleable tasks (tasks whose work increase greatly as the number of allotted processors increases). This algorithm is carefully investigated in Section 3.3.

Turek, et al. gave another 2-approximation algorithm for MPTS in 1992 [36]. Their work has some important contributions. Their algorithm constructs the schedule in two steps. First, the allotment is determined, and then tasks are assigned to processors by an NPTS algorithm. This two-step paradigm preserves the approximation ratio of the NPTS algorithm. In fact, the algorithm of [4] can be considered as a simplified version of this algorithm, so that algorithm is also faster. The time complexity of Turek's algorithm is $O(mnN(m, n))$, where $N(m, n)$ is the time complexity of solving NPTS.

Ludwig and Tiwari's algorithm for MPTS also has an approximation ratio of 2 [30], and it also adopts the two-step paradigm. In the first step, the allotment is determined by searching for the allotment that minimizes:

$$w(p_1, p_2 \dots p_n) = \max\left\{\frac{1}{m}\sum_{j=1}^{n} p_j t_j(p_j), \max_{1\le j\le n}\{t_j(p_j)\}\right\}.$$

It can be proved that such allotment preserves the approximation ratio of the NPTS algorithm. The second step is constructing the schedule from the allotment obtained in the first step by some NPTS algorithm. As a result, the time complexity of this algorithm is reduced to $O(mn + N(m, n))$.

The algorithm by Mounie, et al. is a $\sqrt{3}$-approximation algorithm for MPTS, for cases with at least 7 processors [31]. This algorithm adopts the dual approximation method [19], and finds the solution through a binary search, which requires multiple calls to the dual approximation procedure. Depending on the problem instance, each of the dual approximation procedure may involve either a call to the List algorithm [13], or solving a knapsack problem [25]. As a result, the time complexity of this algorithm is high. In our simulations (see Section 5), we found schedules produced by this algorithm identical to those produced by the LPT algorithm for many cases. This is consistent with our discussion in Section 3.2: for most cases, the makespan produced by LPT is close to the optimal makespan. Their 3/2 approximation algorithm uses the same dual approximation method [32], and is more effective but has higher time complexity.

In 2002, Jansen and Porkolab introduced their polynomial time approximation scheme for MPTS [21]. This scheme is based on linear programming, and requires the number of processors fixed. First, all tasks are divided into two subsets based on their processing times on m processors. One subset contains long tasks, and the other contains short tasks. Second, all the relative schedules of large tasks are constructed and small tasks are incorporated. Finally, the scheme selects the schedule with the minimum makespan. The time complexity of the scheme is linear in the number of tasks. In fact, the time to construct the schedule is large in practice, because each combination of task allotment and task ordering must be tried in the scheme. The total number of such combinations is exponential in the number of processors. Jansen also gave an asymptotic fully polynomial time approximation scheme for MPTS [20]. For a schedule with makespan no more than $(1 + \varepsilon)\ C^*_{max} + s$, the scheme requires $O(n^2(\varepsilon^{-2}+\ln n)\max(\ln\ln(n\varepsilon^{-1}), s\varepsilon^{-2})\ln(\varepsilon^{-1})+\varepsilon^{-2}s\max(M(s), \varepsilon^{-3}\ln(\varepsilon^{-1})))$ time, where $s = O(\log_{1+\varepsilon}(\varepsilon^{-1}))$ and $M(s)$ is the time for inverting an $s \times s$ matrix.

3. Analyses of some existing algorithms

In this section, we give detailed analyses of some existing algorithms. The purpose is to find their specific features so as to guide us in designing algorithms for MPTS. First, we analyze the List [14] and LPT [15] algorithms which were originally designed for the $P\|C_{max}$ problem. We apply them to MPTS not only because of the close relationship between $P\|C_{max}$ and MPTS, but also because they are both simple and efficient algorithms. In fact, for many cases, schedules produced by them for MPTS are of high qualities according to our analyses. Second, we analyze the algorithm introduced in [4,36], and this algorithm will be used as one of the building blocks of our scheduling algorithm OCM.

3.1. Analysis of the list algorithm

The List algorithm [14] introduced by Graham in 1966 is one of the most classic scheduling algorithms, and is also quite effective in many cases. The List algorithm discussed here is for sequential tasks (i.e. for the $P\|C_{max}$ problem):

Given a list of tasks in arbitrary order, whenever a processor becomes idle, select the first task, delete it from the list, and start processing it immediately on the processor. Continue this process until the list is empty.

For the sake of clarity, we refer to this algorithm as List_seq, and the List algorithm for NPTS [13] as List_par. To apply List_seq to MPTS, we only need to set the allotment to $(1, 1 \dots 1)^T$. Since the indices of tasks can be set arbitrarily, we assume $t_1(1) \ge t_2(1) \ge \cdots \ge t_n(1)$ for the following discussions. The lemma below characterizes the performance of List_seq when applied to MPTS (Please note that part of this proof is inspired by the analysis of algorithms for $P\|C_{max}$ [17]).

Lemma 1. *Let T_k be a critical task and C^L_{max} be the makespan produced by List_seq. The following relation holds:*

$$C^L_{max} \le C^*_{max} + \left(1 - \frac{1}{m}\right) t_k(1).$$

Proof. According to the List_seq algorithm, no processor can be idle before time $C^L_{max} - t_k(1)$, otherwise T_k would have been started earlier on that processor. The total amount of work completed at that time is at most $\sum_{j=1}^{n} t_j(1) - t_k(1)$, so we have:

$$m\left(C^L_{max} - t_k(1)\right) \le \sum_{j=1}^{n} t_j(1) - t_k(1).$$

This is equivalent to:

$$C^L_{max} \le \frac{1}{m}\sum_{j=1}^{n} t_j(1) + \left(1 - \frac{1}{m}\right) t_k(1). \tag{3.1}$$

We turn to the analysis of an optimal schedule. The optimal makespan has the following lower bound:

$$C^*_{max} \ge \frac{1}{m}\sum_{j=1}^{n} p^*_j t_j(p^*_j) \tag{3.2}$$

since the expression on the right side represents the ideal case where all processors share equal amount of work. By observing $p^*_j \ge 1 (j = 1, 2 \dots n)$ and the increasing work assumption, we have $\frac{1}{m}\sum_{j=1}^{n} p^*_j t_j(p^*_j) \ge \frac{1}{m}\sum_{j=1}^{n} t_j(1)$. Combining this with (3.2) yields:

$$C^*_{max} \ge \frac{1}{m}\sum_{j=1}^{n} t_j(1). \tag{3.3}$$

By combining (3.1) and (3.3), we get

$$C^L_{\max} \le C^*_{\max} + \left(1 - \frac{1}{m}\right) t_k(1). \quad \square$$

From Lemma 1 it can be noticed that the approximation ratio of List_seq for MPTS depends on the critical task. Lemma 2 makes this relation clearer.

Lemma 2. *In the schedule produced by List_seq, if T_k is a critical task, then $C^L_{\max} \le \left(1 + \frac{m-1}{k}\right) C^*_{\max}$.*

Proof. According to our assumption, we have $t_1(1) \ge t_2(1) \ge \cdots \ge t_n(1)$, so there are at least k tasks with sequential processing time no less than $t_k(1)$, and the total work of these tasks is at least $kt_k(1)$. For any valid schedule, there must be one processor that carries out at least $kt_k(1)/m$ of this work, which implies:

$$C^*_{\max} \ge \frac{k}{m} t_k(1).$$

By applying this to Lemma 1, the following relation can be obtained:

$$C^L_{\max} \le C^*_{\max} + \left(1 - \frac{1}{m}\right) \frac{m}{k} C^*_{\max} = \left(1 + \frac{m-1}{k}\right) C^*_{\max}. \quad \square$$

By Lemma 2 the following conclusions can be drawn. Generally, the best case of List_seq occurs when the critical task has the shortest sequential processing time ($k = n$), and the worst case occurs when the critical task has the longest sequential processing time ($k = 1$). For the latter case, the makespan produced by List_seq is within m times the optimal makespan, so we have the following theorem.

Theorem 1. *The List_seq algorithm is an m-approximation algorithm for MPTS.*

It should be noted that this approximation ratio (m) is tight, as shown below by Instance 1. The makespan produced by List_seq is M, while the optimal schedule allots m processors to T_1, and one processor to each of the remaining tasks. This leads to the makespan of $M/m + 1$. Their ratio $M/(M/m + 1)$ approaches m when M approaches infinity. Finally, List_seq can be implemented with time complexity $O(n \log m)$ by maintaining the processor set as a heap.

Instance 1. Processing times of tasks are $t_1(p) = M/p$ (M is a large real number), $t_j(p) = 1 (j = 2, 3 \ldots n) \cdot m = n$. Tasks in the list are in arbitrary order.

3.2. Analysis of the LPT algorithm

The LPT algorithm [15] was also introduced by Graham, as a special case of List_seq. The only difference between LPT and List_seq is that for LPT, tasks of the list are in non-increasing order of the sequential processing time. In other words, the task list is $L = (T_1, T_2 \ldots T_n)$ (recall that $t_1(1) \ge t_2(1) \ge \cdots \ge t_n(1)$ according to our assumption). Graham proved that for the problem with sequential tasks ($P\|C_{\max}$), the approximation ratio can be reduced from 2 to 4/3 by this ordering preprocess. But that is not the case for MPTS as seen from the following analysis.

Theorem 2. *LPT is an m-approximation algorithm for MPTS, and this approximation ratio is tight.*

Proof. Proofs of Lemmas 1 and 2 also apply to the LPT algorithm, because the order of the task list is not taken into consideration when proving these lemmas. From these lemmas, we conclude that LPT is an m-approximation algorithm. This approximation ratio (m) is tight, as can be verified by Instance 1. $\square$

Although LPT has the same approximation ratio as that of List_seq, it may be much more effective in practice. This can be illustrated by Lemma 1. If tasks are in non-increasing order of the sequential processing time, there is a much greater chance that the critical task has a small sequential processing time, because small tasks with large indices appear in the latter part of the list. In that case, $t_k(1)$ would be insignificant compared with $C^*_{\max}$, and according to Lemma 1, the schedule produced by LPT would be close to the optimum. Additionally, in many practical scenarios, the number of tasks is far greater than the number of processors. For those cases, the value of k in Lemma 2 is close to n, so the value of $(m-1)/k$ is trivial. This is another reason why schedules produced by LPT are closer to the optimum. In reality, LPT performs much better than its worst case behavior, as revealed by the following corollary.

Corollary 1. *For the schedule produced by LPT, if each processor is assigned at least two tasks, the makespan is within twice of the optimal makespan.*

Proof. According to the hypothesis, the critical task T_k cannot start at time 0. In the schedule, there are m tasks starting at time 0. These tasks must precede T_k in the list, because otherwise, T_k would have been dispatched earlier than them, and started at time 0. This means that $k \ge m + 1$. By Lemma 2, we have

$$C^{LPT}_{\max} \le \left(1 + \frac{m-1}{k}\right) C^*_{\max} \le \left(1 + \frac{m-1}{m+1}\right) C^*_{\max} \le 2C^*_{\max}. \quad \square$$

Our simulations in Section 5 are consistent with the result of Corollary 1. For many cases, the results of LPT are dramatically better than those produced by some other approximation algorithms (like [36,30], and so on). In addition, schedules produced by some approximation algorithms (like [4,31]) are identical to that of LPT.

The time complexity of LPT is the time for sorting tasks plus the time for dispatching tasks. That is $O(n \log n) + O(n \log m) = O(n \log mn)$.

3.3. Analysis of the OM algorithm

This section describes an optimization algorithm for MPTS. It is one of the building blocks of our scheduling algorithm OCM. Since it deals with tasks that appear in the middle part of the schedule, we call it Optimizing the Middle algorithm (OM). The OM algorithm was first introduced in [4,36]. We view it from another perspective, and give some minor modifications to it. For the sake of clarity, we give an additional definition: a task is called a singular task of the schedule, if it is a critical task, and the only task of its allotted processor(s).

The OM algorithm starts from the schedule produced by LPT, and proceeds through an iterative process. For each round of the iteration, a new allotment is generated by increasing the allotment of the first (the first that appears in the task list) singular task by one, while preserving the allotments of other tasks. Once the allotment is determined, the task list is constructed as $L = (L_p, L_s)$, where L_p is the list constructed by parallel tasks, and L_s is made up of sequential tasks. Tasks in L_s are in non-increasing order of the sequential processing time. The last step of each round of iteration is to construct the schedule from list L by the List_par algorithm. The iterative process terminates if one of the following conditions is satisfied:

(1) The sum of allotments for parallel tasks exceeds m.
(2) There is no singular task in the schedule.
(3) The makespan is greater than that of the previous round.

It should be noted that if condition (3) is satisfied, the final schedule must be restored to the one produced in the previous round.

It has been proved that the makespan produced by OM is within twice of the optimal makespan [4,36]. Besides, this approximation ratio of 2 is tight, as illustrated by Instance 2. For this instance, the makespan produced by OM is 1, while the optimal schedule allots each task m processors, yielding a makespan of $M/(2M-1)$. Their ratio $(2M-1)/M$ approaches 2, when M approaches infinity. Finally, OM can be implemented in time $O(n \log n + mn \log m)$ [4].

Instance 2. $n = M$ (M is a large integer) and $m = 2M - 1$. Processing times of tasks are $t_j(p_j) = 1/p_j (j = 1, 2 \ldots n)$.

4. Descriptions of our algorithms

In this section, we introduce our algorithms for MPTS, and give detailed analyses. In particular, we separate a schedule into 3 parts: top, middle and bottom, and employ respectively OT, OM, and OB algorithms to optimize these parts. Besides these basic building blocks of OCM, we also design other algorithms and techniques to further optimize the performance and efficiency of OCM. This section is organized as follows: the OT optimization algorithm is introduced in Section 4.1, which is capable of reducing the approximation ratio from linear to logarithmic in the number of processors. The OB algorithm described in Section 4.2 divides the task set into subsets of highly and poorly malleable tasks (Generally, a task is considered to be highly malleable if its speedup on multiple processors is high, and otherwise, it is considered to be poorly malleable), and processes them separately. The algorithm introduced in Section 4.3 (Multi_LayerS) further reduces the makespan of highly malleable tasks. The sorting strategy introduced in Section 4.4 is capable of reducing the total number of sorting processes from $3n + 1$ to 1. Section 4.5 defines a rule (Rule 1) which makes the algorithm much faster. A formal description of the OCM algorithm is given in Section 4.6. For clarity, the basic structure of the OCM algorithm is illustrated by Fig. 1.

4.1. The OT optimization algorithm

Although LPT is effective in most cases (according to Corollary 1), its worst case performance is unacceptable. If we analyze the structure of Instance 1, several reasons for the poor performance of LPT can be identified. First, LPT does not make use of the flexibility that each task can be allotted more than one processor. Second, scheduling of the critical task requires careful consideration, because it directly decides the makespan. However, LPT fails to give any special attention to it.

In fact, only a simple optimization algorithm suffices for reducing the approximation ratio of LPT. We call it Optimizing the Top (OT), since it is specifically designed for the critical task (which appears at the top of the schedule graph, see Fig. 2). Suppose the finishing time of machine M_i is $f_i (i = 1, 2 \ldots m)$ in the schedule produced by LPT. Without loss of generality, we assume $f_1 \leq f_2 \leq \cdots \leq f_m$, and the critical task appears on machine M_m.

The problems of LPT are addressed by OT. For the critical task T_k, there are m choices: it can start at time $f_m - t_k(1)$ on one processors (M_m), or start at time f_1 on two processors (M_1, and M_m), or start at time f_2 on three processors (M_1, M_2, and M_m), and so on. The OT algorithm selects the case that minimizes the makespan, and schedules T_k accordingly. The makespan after adopting the OT algorithm is then:

$$C_{\max}^{OT} = \min\left\{f_m, \min_{1\leq i\leq m-1}\{f_i + t_k(i+1)\}\right\}. \tag{4.1}$$

Theorem 3. *The List_seq algorithm optimized by the OT algorithm is a $(1 + \ln m)$-approximation algorithm for MPTS.*

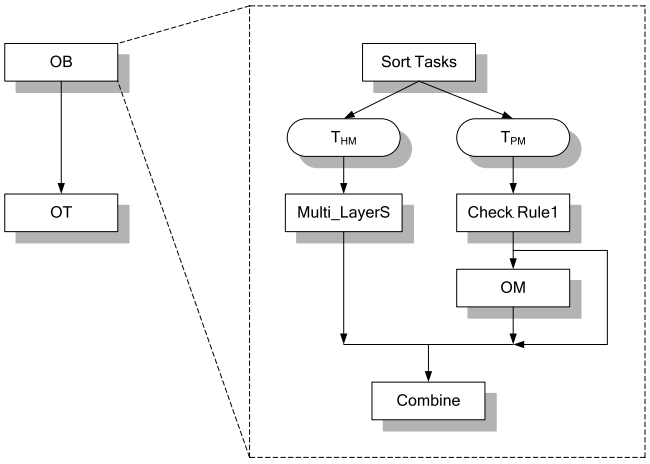

Fig. 1. The basic structure of the OCM algorithm.

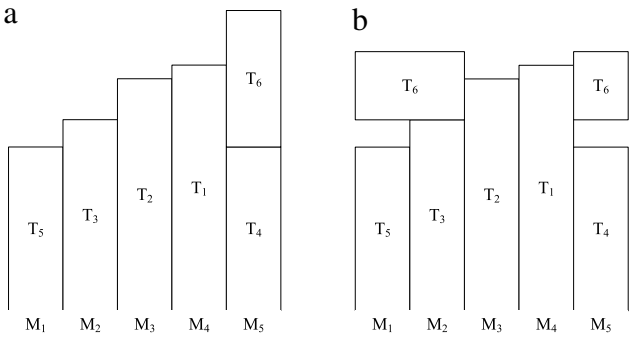

Fig. 2. Optimizing the top. (a) is the schedule produced by LPT, and (b) is processed by OT.

Proof. Formula (4.1) gives rise to a series of inequalities:

$C_{\max}^{OT} \leq f_m$
$C_{\max}^{OT} \leq f_1 + t_k(2)$
$\ldots$
$C_{\max}^{OT} \leq f_{m-1} + t_k(m).$

Combining these inequalities, we get:

$$C_{\max}^{OT} \leq \frac{1}{m}\sum_{i=1}^{m} f_i + \frac{1}{m}\sum_{i=2}^{m} t_k(i). \tag{4.2}$$

According to the List_seq algorithm, no processor may be idle before its finishing time, so we have $\sum_{i=1}^{m} f_i = \sum_{j=1}^{n} t_j(1)$, and by (3.3), we get:

$$C_{\max}^{*} \geq \frac{1}{m}\sum_{i=1}^{m} f_i. \tag{4.3}$$

From the increasing work assumption, we have:

$$\frac{1}{m}t_k(i) \leq \frac{1}{i}t_k(m) \leq \frac{1}{i}C_{\max}^{*}. \tag{4.4}$$

By applying (4.3) and (4.4) to (4.2), the following inequality can be obtained:

$$C_{\max}^{OT} \leq C_{\max}^{*}\sum_{i=1}^{m}\frac{1}{i} \leq (1 + \ln m)C_{\max}^{*}. \quad \square$$

It can be easily verified that the schedule produced by List_seq with OT for Instance 1 is the optimal schedule. In practice, the OT

algorithm can be applied several rounds on different critical tasks. Clearly, the time complexity for one round of OT is $O(m)$. Finally, the OT algorithm may also be applied to other algorithms except LPT.

4.2. The OB optimization algorithm

In our simulations in Section 5, we find the OM algorithm effective enough for most cases. However, results produced by OM were much worse for cases with highly malleable tasks. For this type of tasks, speedups are close to the numbers of processors allotted to them, or equivalently, the work increases mildly as the number of processors increases. Therefore, allotting large numbers of processors to such tasks should be encouraged. But OM generally allots a small number of processors for each task. To see this, please note that each task is allotted only one processor at the beginning of OM, and for each round of the iteration, one task's allotment is increased by one. The number of iterations for OM is generally small, because the conditions for terminating the algorithm (see Section 3.3) are easily satisfied in most cases.

This is the reason for the poor performance of OM when applied to cases with highly malleable tasks. Instance 2 is such a case, the tasks of which are perfectly malleable. In practice, many applications, like matrix multiplication, Jacobi iteration, and many genetic algorithms, do not involve too much communication or synchronization when running on multiple processors, and they are good examples of highly malleable tasks. Moreover, these applications are frequently encountered in practice, so it is significant to give special attention to this kind of tasks in order to design effective algorithms for real applications.

It is generally straightforward to deal with highly malleable tasks. As indicated by Instance 2, the schedule obtained by allotting m processors to each task is likely to be better than that produced by OM, or by other algorithms for MPTS. This is not a coincidence. For many highly malleable tasks, the makespan of the above schedule is very close to the optimum. Based on this observation, we introduce a simple optimization algorithm that is effective for both highly and poorly malleable tasks. The algorithm consists of three steps. First, the task set is divided into two subsets: one contains poorly malleable tasks, and the other contains highly malleable tasks. Suppose $(\pi(1), \pi(2) \ldots \pi(n))$ is an arbitrary permutation of integers in $(1, 2 \ldots n)$, we define these two subsets as $T_{PM} = \{T_{\pi(1)}, T_{\pi(2)} \ldots T_{\pi(k)}\}$ and $T_{HM} = \{T_{\pi(k+1)}, T_{\pi(k+2)} \ldots T_{\pi(n)}\}$, respectively, where k is an integer between 1 and n. Second, tasks in T_{PM} are scheduled by the OM algorithm, and tasks in T_{HM} are each allotted m processors. Finally, the two partial schedules are combined. In our implementation, the tasks in T_{HM} are scheduled first (at the bottom of the schedule graph, see Fig. 3), so we call this algorithm Optimizing the Bottom (OB).

Let $C^{\mathrm{OM}}_{\max}(k)$ denotes the makespan produced by OM for task set $\{T_{\pi(1)}, T_{\pi(2)} \ldots T_{\pi(k)}\}$. The makespan of OB can be represented as:

$$C^{OB}_{\max} = C^{\mathrm{OM}}_{\max}(k) + \sum_{j=k+1}^{n} t_{\pi(j)}(m). \tag{4.5}$$

Therefore, to implement OB, two factors remain to be determined. One is the permutation of tasks (π), and the other is the starting index for highly malleable tasks (k). The value of k can be obtained by trying all integers from 1 to n, and taking the value corresponding to the minimal makespan, that is:

$$C^{OB}_{\max} = \min_{1 \le k \le n} \left\{ C^{\mathrm{OM}}_{\max}(k) + \sum_{j=k+1}^{n} t_{\pi(j)}(m) \right\}. \tag{4.6}$$

To determine the task permutation, it is desirable to sort tasks by some measure of their malleability, so that highly malleable tasks are categorized into T_{HM}, and poorly malleable tasks are categorized into T_{PM}. Detailed method and analysis for sorting tasks will be discussed in Section 4.4.

The OB algorithm involves calling the OM algorithm n times with different parameters (In fact, some calls can be skipped, as illustrated in Section 4.5). The total time complexity is:

$$O\left(\sum_{k=1}^{n}(k \log k + mk \log m)\right) = O(n^2 \log n + mn^2 \log m).$$

This time complexity is not low. Moreover, the time for sorting tasks and constructing the schedule for tasks in T_{HM} are not included, which are also large. To improve the effectiveness as well as speed of OM, we devise some algorithms and improving techniques, as described in the following sections.

4.3. Reducing the makespan of highly malleable tasks

According to the description of OB, the makespan produced by tasks in T_{HM} is $\sum_{j=k+1}^{n} t_{\pi(j)}(m)$, as each task is allotted m processors. More often than not, this makespan can be easily reduced through a simple algorithm. This is especially true when m is large: although tasks in T_{HM} are considered as highly malleable and suitable for relatively large allotments, m is too large for them.

To describe our algorithm to reduce the makespan of highly malleable tasks, we define $a_j(\tau)$ as the minimum allotment for T_j such that its processing time is not greater than τ. That is:

$$a_j(\tau) = \begin{cases} \min\{p_j \mid t_j(p_j) \le \tau\} & t_j(m) \le \tau \\ m+1 & t_j(m) > \tau. \end{cases}$$

For a given target value τ, we try to construct a multi-layered schedule for tasks in T_{HM}. If the makespan of the schedule is not greater than τ, we mark this target value as valid. Otherwise, the target value is marked as invalid. As a result, our goal is to find the minimal valid target value. We do this through a binary search, with the initial upper bound $\sum_{j=k+1}^{n} t_{\pi(j)}(m)$ and lower bound $\frac{1}{m}\sum_{j=k+1}^{n} t_{\pi(j)}(1)$. Given a target value τ, the specific algorithm to construct the multi-layered schedule is given below:

```
Multi_LayerS(τ, k)
1    s = 0; avp = m; t = τ;
2    for i = k + 1 to n
3    do
4      if (a_π(i)(τ) ≤ avp)then
5        Assign a_π(i)(τ)processors to T_π(i), and start it at time s.
6        avp = avp − a_π(i)(τ)
7      else if (a_π(i)(τ) > m)
8        return false
9      else   //Start a new layer
10       Set c to the largest processing time of all tasks in the current layer.
11       s = s + c; τ = τ − c;   avp = m;   i = i − 1
12       if (τ ≤ 0)return false endif
13     endif
14   endfor
15   return true
```

Inputs of the algorithm are the target value (τ) and starting index of highly malleable tasks (k). If target value τ is valid, the algorithm returns true, and a schedule with makespan not greater than τ is created; otherwise, it returns false. Variable s represents the starting time of the current layer. avp is the number of available processors in the current layer. Each task T_j is allotted $a_j(\tau)$ processors and started at time s, if $a_j(\tau)$ is not greater than m. If there are not enough processors, a new layer is initiated, with starting time equal to the largest finishing time of tasks in the

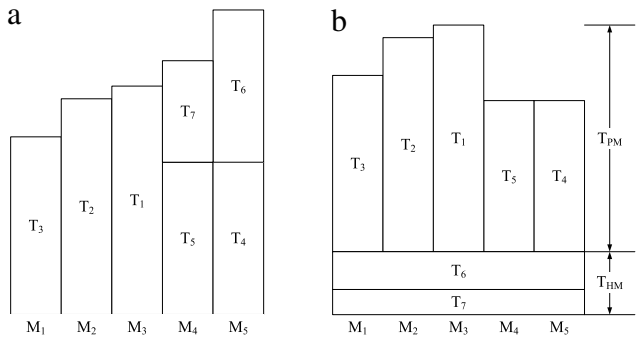

Fig. 3. Optimizing the Bottom. (a) is the schedule produced by OM, and (b) is the result of OB.

current layer. In addition, the value of avp is restored to m and the target value is reduced accordingly.

It should be noted that if one task in the current layer is allotted more than one processor (which is often the case), the target value for the next layer is at most half of the one for the current layer. To see this, suppose the target value for the current layer is τ, and task T_j in the current layer is allotted $a_j(\tau) > 1$ processors. According to the definition of function $a_j(\tau)$, we have $t_j(a_j(\tau)) \leq \tau$ and $t_j(a_j(\tau) - 1) > \tau$. From the increasing work assumption, we have

$$t_j(a_j(\tau) - 1) \leq \frac{a_j(\tau)}{a_j(\tau) - 1} t_j(a_j(\tau)) \leq 2t_j(a_j(\tau)).$$

We claim that $t_j(a_j(\tau)) > \tau/2$. This is true because otherwise, we would get $t_j(a_j(\tau) - 1) \leq \tau$ from the formula above, which contradicts our definition of function $a_j(\tau)$. Therefore, the target value for the next layer is at most $\tau - t_j(a_j(\tau)) < \tau/2$.

The point here is that the amount of available resources for tasks becomes smaller and smaller as the algorithm proceeds, so the ordering of tasks should be carefully considered so as to minimize the makespan of *Multi_LayerS*. Generally, long tasks should be scheduled with large target values at bottom layers, and short tasks should be scheduled with small target values at top layers. As a result, sorting tasks by the non-increasing order of the sequential processing time is a necessity, and this is consistent with our sorting strategy described in Section 4.4.

Obtaining the allotment of each task requires $O(\log m)$ time through the binary search, so the time complexity of *Multi_LayerS* is $O((n-k)\log m)$. The time to find the schedule with the minimal valid target value is:

$$O\left(\log \frac{m \sum_{j=k+1}^{n} t_{\pi(j)}(m)}{\sum_{j=k+1}^{n} t_{\pi(j)}(1)} (n-k)\log m\right) = O((n-k)\log^2 m),$$

which is larger than that of the original OB algorithm ($O(n-k)$). For most cases, there is no need to wait until the binary search converges. A few rounds of *Multi_LayerS* would dramatically reduce the makespan. Thus, the time complexity of constructing the schedule for tasks in T_{HM} is $O((n-k)\log m)$.

4.4. Improving the sorting strategy

To determine the permutation of tasks, a certain measure of the malleability of tasks should be employed. This is reasonable. Through extensive simulations, we find this idea effective but slow. In fact, for each selection of the starting index of highly malleable tasks (k), four sorting processes are involved: (1) Tasks are sorted by their malleability and divided into subsets of highly malleable and poorly malleable tasks. (2) In the execution process of OM, tasks are sorted by their allotments, and divided into subsets of parallel and sequential tasks. (3) Sequential tasks are sorted by their processing times in OM. (4) Highly malleable tasks are sorted in *Multi_LayerS* to reduce the makespan. Three of these sorting processes are repeated for each value of k, because task subsets are different for different values of k. Therefore, the total number of sorting processes is $3n + 1$.

In practice, we find that by sorting tasks in non-increasing order of the sequential processing time, the OB algorithm performs fairly well. As a result, we adopt this sorting strategy. For the OB algorithm, tasks in subset T_{HM} are allotted relatively large numbers of processors. According to the increasing work assumption, extra work will be incurred by them. In order to minimize the amount of extra work, it is desirable that this subset contain short tasks, because the extra work incurred by a short task is also trivial. According to OB, tasks in T_{HM} are selected from the latter part of the permutation, which are short if tasks are in non-increasing order of the sequential processing time. This is the first reason for the effectiveness of our sorting strategy.

To understand the second reason, please refer to (4.6). The makespan of OB is the sum of two terms, which are both functions of the same variable (k). If we define

$$f(k) = C_{\max}^{\mathrm{OM}}(k)$$

$$g(k) = \sum_{j=k+1}^{n} t_{\pi(j)}(m),$$

then $f(k)$ is a monotonically increasing function of k, while $g(k)$ is a monotonically decreasing function of k. We can extend these discrete functions to continuous functions (by interpolation, for example). It can be noticed that if tasks are in non-increasing order of the processing time, the continuous function $g(x)$ is a convex function, and the sum function $(f(x) + g(x))$ is more likely to be convex (Fig. 4(a)). In that case, the minimum of the sum function is obtained in the middle between 1 and n. In other words, the makespan produced by OB is smaller than that of OM ($f(n)$). If we employ the opposite approach, where tasks are sorted in non-decreasing order of the processing time, then $g(x)$ is a concave function, and more often than not, function $g(x) + f(x)$ is also a concave function (Fig. 4(b)), so the minimum is obtained at the boundary point (n), which means the schedule produced by OB is identical to that of OM.

The third reason for the effectiveness of our sorting strategy has been discussed in Section 4.3. Through our novel sorting strategy, the first sorting process (by the malleability) is no longer needed. Besides, it is not necessary to repeat the third (sorting by the sequential processing time in OM) and fourth (sorting highly malleable tasks by *Multi_LayerS*) sorting processes for each value of k, because both of them require that tasks should be in non-increasing order of the sequential processing time, which is consistent with our sorting strategy. In fact, the second sorting process (by the allotment) is also unnecessary. To see this, please recall that for each round of OM, parallel tasks are scheduled first, and then sequential tasks. So we only need to prove that after sorting tasks by the sequential processing time, all parallel tasks appear in the former part of the permutation. Or equivalently, we only need to prove that the sequential processing time of each parallel task is greater than or equal to those of sequential tasks.

Theorem 4. *Suppose tasks in the list $(T_1, T_2 \ldots T_n)$ are in non-increasing order of the sequential processing time. In the schedule produced by OM, if T_{j1} is a parallel task and T_{j2} is a sequential task, then $j_1 < j_2$.*

Proof. According to the OM algorithm, T_{j1} is a sequential task at the beginning. It becomes a parallel task because it is a singular

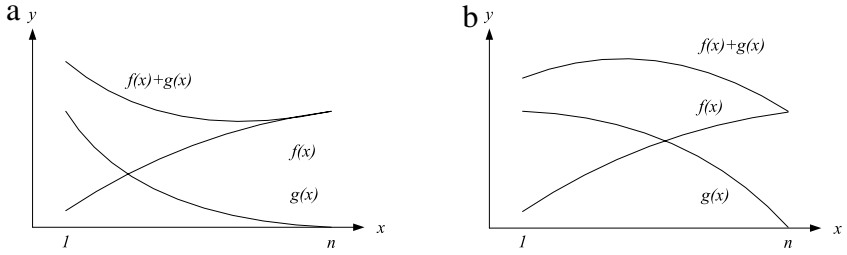

Fig. 4. For (a), $g(x)$ is a convex function, and for (b), $g(x)$ is a concave function.

task at some point during the process of OM, and T_{j2} is also a sequential task at that point. To see this, please note that there is only one case where the allotment of a task can be decreased. In that case, the schedule of the previous round is restored and the algorithm terminates. If T_{j2} were not a sequential task before T_{j1} became a parallel task, the algorithm would have terminated before T_{j1} became a parallel task, which contradicts our assumption that T_{j1} end up as a parallel task.

At the point when T_{j1} is a (sequential) singular task, $t_{j1}(1)$ is equal to the makespan of the schedule, which is not less than $t_{j2}(1)$. So $t_{j1}(1) \geq t_{j2}(1)$.

If $t_{j1}(1) > t_{j2}(1), j_1$ is clearly smaller than j_2 by our sorting hypothesis. Otherwise, T_{j1} and T_{j2} are both singular tasks at that time. In that case, T_{j1} also appears before T_{j2} in the list because it becomes a parallel task prior to T_{j2}. □

Finally, it should be pointed out that because OM selects the first singular task in the list, the order of tasks for different starting indices of highly malleable tasks (k) remains the same. Therefore, it requires only one sorting process for the whole process of OB. The total number of sorting processes is reduced from $3n + 1$ to 1 by our sorting strategy.

4.5. Reducing the number of calls to OM

As discussed in Section 4.2, the OB algorithm requires values of $C^{\mathrm{OM}}_{\max}(k)(k = 1, 2 \ldots n)$. This involves n calls to OM, which accounts for a significant amount of time. By carefully investigating the OM algorithm, we find some calls unnecessary. For example, to calculate the value of $C^{\mathrm{OM}}_{\max}(k)$, a schedule for the task set $\{T_{\pi(1)}, T_{\pi(2)} \ldots T_{\pi(k)}\}$ is constructed. According to OM, task $T_{\pi(j)}$ $(j < k)$ is processed before tasks $T_{\pi(j+1)}, T_{\pi(j+2)} \ldots T_{\pi(k)}$ by OM. At the time after $T_{\pi(j)}$ is scheduled and before $T_{\pi(j+1)}$ is scheduled, the only tasks contained in the schedule are $T_{\pi(1)}, T_{\pi(2)} \ldots T_{\pi(j)}$, so it is very likely that the finishing time of $T_{\pi(j)}$is equal to the makespan produced by OM for task set $\{T_{\pi(1)}, T_{\pi(2)} \ldots T_{\pi(j)}\}$ namely $C^{\mathrm{OM}}_{\max}(j)$. If that is the case, the call to OM for the value of j can be bypassed. The following rule shows when this happens.

Rule 1. During the execution process of OM for task set $\{T_{\pi(1)}, T_{\pi(2)} \ldots T_{\pi(k)}\}$, if the finishing time of task $T_{\pi(j)}(j < k)$ is the largest before $T_{\pi(j+1)}$ is scheduled, and if one of the following conditions is satisfied at that time, then the finishing time of $T_{\pi(j)}$ is equal to $C^{\mathrm{OM}}_{\max}(j)$.

(1) The sum of allotments for parallel tasks is m.
(2) There is no singular task in the schedule.
(3) There is a singular task, but the sum of allotments for parallel tasks is $m - 1$.

The correctness of Rule 1 can be easily verified by inspecting the details of OM. Incorporating Rule 1 into OB does not incur too much extra time. In fact, Rule 1 can be checked in time O(1) if implemented properly. It can be seen that conditions of Rule 1 are more likely to be satisfied when the value of j is large, because in that case, the sum of allotments for parallel tasks and the number of tasks for each processor are relatively large. This is another favorable factor for reducing the time of OB, since the time complexity of OM is high when j is large. The above analysis has been confirmed by our simulations in Section 5.

Owning to Rule 1, OM is called by OB in descending order of the starting index of highly malleable tasks. That is, OM is first called with argument n to obtain the value of $C^{\mathrm{OM}}_{\max}(n)$. Second, if the value of $C^{\mathrm{OM}}_{\max}(n-1)$ is not determined according to Rule 1, OM is called with argument $n - 1$, otherwise, the call to OM is skipped. This process continues until the value of $C^{\mathrm{OM}}_{\max}(1)$ is determined. Our simulations in Section 5.5 revealed that by applying Rule 1 to OB, the number of calls to OM can be dramatically reduced, which makes the OB algorithm much faster.

4.6. The OCM scheduling algorithm

So far, we have finished the descriptions of all optimization algorithms and improving techniques. Combining them gives rise to our novel scheduling algorithm for MPTS, we name it OCM (Optimizations Combined for Mpts). The basic structure of OCM has been given in Fig. 1: First, a schedule is constructed by the OB algorithm, and then the OT algorithm is applied to optimize critical tasks. Details of the OB algorithm are illustrated in the right part of Fig. 1. At first, all tasks are sorted according to the sorting strategy described in Section 4.4. After that, subsets of highly and poorly malleable tasks go through separate routines. Highly malleable tasks are processed by the *Multi_LayerS* algorithm to produce a multi-layered partial schedule. On the other hand, if the call to the OM algorithm cannot be skipped according to Rule 1 in Section 4.5, OM is called to produce a partial schedule for poorly malleable tasks; otherwise, the partial schedule for poorly malleable tasks has already been generated. Finally, the two partial schedules for highly and poorly malleable tasks are combined to produce the schedule of OB.

OCM is trivially a 2-approximation for MPTS, because OM is a 2-approximation algorithm, the makespan produced by OB is not greater than that of OM, and the OT algorithm does not increase the makespan produced by OB.

5. Simulation results

Extensive simulations have been carried out in order to examine the effectiveness of OCM. In the simulations, OCM and some existing algorithms are implemented in C++ language on Linux platform, and executed for a comparative study. In particular, two classes of simulations were conducted: One was on randomly generated datasets and the other was on the standard benchmark suite SPLASH-2 [41].

5.1. Construction of random datasets

A large number of random datasets with a wide range of parameters were used in our simulations. Theoretically, the processing time of each task can be any positive real number.

However, in practice, if the processing time of one task is too much larger than that of another one, the smaller task can be ignored, because it cannot have any essential impact on the makespan. To avoid such case, the sequential processing time of each task is restricted to be a random number uniformly distributed between 10 and 100. Each dataset is characterized by 3 parameters: the sequential ratio, the task/processor ratio and the number of processors.

According to Amdahl's law [1], the processing time of each task can be divided into two parts: the part that can only be executed sequentially (s), and the part that can be executed in parallel (p). Its processing time on m processors is therefore $s + p/m$. Clearly, this model satisfies the decreasing time and increasing work assumptions, and our datasets were generated in accordance with this model. We define the sequential ratio of the task as $s/(s + p)$. It characterizes the weight of the sequential part of the task. Generally, computation intensive tasks tend to have small sequential ratios, while memory intensive tasks tend to have large sequential ratios. The task/processor ratio of a dataset refers to the ratio between the numbers of tasks and processors.

Besides OCM, we also implemented several other practical algorithms so as to compare their performance. These practical algorithms include: the LPT algorithm [15], the 2-approximation algorithm introduced by Banerjee in 1990 [4], the 2-approximation algorithm introduced by Turek et al. in 1992 [36], the 2-approximation algorithm introduced by Ludwig and Tiwari in 1994 [30], and the $3^{1/2}$-approximation algorithm introduced by Mounié et al. in 1999 [31]. In the following discussion, these algorithms are referred to as LPT, Ba90, TWY92, LT94, and MRT99, respectively.

A natural way of evaluating an approximation algorithm is by the approximation ratio. However, obtaining the optimum for an MPTS instance is not easy, due to the NP-hardness of MPTS. As a result, we use the effectiveness ratio (ER) instead:

$$ER(A) = \frac{C^A_{\max} - C_{\mathrm{LB}}}{C_{\mathrm{LB}}} \times 100\%$$

$$C_{\mathrm{LB}} = \frac{1}{m}\sum_{j=1}^{n} p_j^b t_j(p_j^b).$$

$C^A_{\max}$ is the makespan produced by Algorithm A, and p_j^b represents the allotment obtained by algorithm [4] for task T_j. According to Lemma 2 of [4], we have $p_j^b \le p_j^*(j = 1, 2 \ldots n)$. By the increasing work assumption and (3.2), the following relations hold:

$$C_{\mathrm{LB}} = \frac{1}{m}\sum_{j=1}^{n} p_j^b t_j(p_j^b) \le \frac{1}{m}\sum_{j=1}^{n} p_j^* t_j(p_j^*) \le C^*_{\max}.$$

Therefore, C_{LB} is a lower bound of the optimal makespan. In this way, the approximation ratio can be estimated to be no more than $ER(A) + 1$.

5.2. Effectiveness with different sequential ratios

The effectiveness ratios of different algorithms with different sequential ratios are listed in Table 1. The sequential ratios range from 0.05 to 0.95. For each selected value r of the sequential ratio, the actual sequential ratio of each task is not necessarily equal to r, but uniformly distributed between $r - 0.02$ and $r + 0.02$. For each category of the sequential ratio, the task/processor ratio might be 1, 2 or 3 (results for larger values of task/processor ratios are discussed in the next section), and the number of processors may be 4, 8, 16, 32 or 64. Besides, 10 datasets were generated for each of the parameter combinations. In summary, for each selected sequential ratio, 150 random datasets were used in our simulations, and each figure in Table 1 is the average obtained from them.

Table 1
Effectiveness ratios (%) with different sequential ratios.

SEQ-RATIO	LPT	BA90	TWY92	LT94	MRT99	OCM
0.05	26.94	13.35	17.16	18.05	13.60	6.74
0.15	25.02	13.06	24.98	19.49	17.53	9.62
0.25	22.69	11.18	27.83	19.31	20.70	10.31
0.35	22.26	10.03	30.24	16.98	21.00	9.48
0.45	18.97	9.74	30.04	16.51	19.76	9.47
0.55	17.54	9.40	30.00	14.78	19.80	9.20
0.65	15.76	7.97	30.06	12.31	17.94	7.93
0.75	13.36	8.00	29.38	11.07	20.35	7.96
0.85	11.98	8.17	28.47	10.40	21.52	8.14
0.95	9.465	8.18	27.80	10.76	25.88	8.18

Table 2
Effectiveness ratios (%) with different task/processor ratios.

T/P-RATIO	LPT	BA90	TWY92	LT94	MRT99	OCM
1	36.77	17.38	21.88	30.20	56.73	12.96
2	7.46	7.25	33.35	7.34	8.32	6.80
3	5.18	5.18	22.54	5.18	5.18	4.87
4	2.34	2.34	17.51	2.34	2.34	2.17
5	1.85	1.85	14.58	1.85	1.85	1.75

The best algorithm for all cases was OCM. The improvements of OCM are greater when the sequential ratio was low. This is consistent with our analysis in Section 4.2 that the OB algorithm is effective for highly malleable tasks (tasks with high speedups). For all cases, the effectiveness ratios produced by OCM are less than 11%, which indicates that our algorithm works well for tasks with various sequential ratios. Also, it should be noted that results produced by LPT are also good when the sequential ratio is high, although it is still not as good as ours. This makes sense, because the LPT algorithm was originally designed for $P\|C_{\max}$, where each task was assigned exactly one processor. A task is likely to be allotted one processor in MPTS when it has a high sequential ratio.

5.3. Effectiveness with different task/processor ratios

Table 2 gives the effectiveness ratios with task/processor ratios ranging from 1 to 5. This is enough to give the trend for cases with larger task/processor ratios (From Table 2 and the following discussions, it can be seen that the effectiveness ratios and the difference between algorithms both become smaller as the task/processor ratio increases.)

For each selected task/processor ratio, three types of datasets were used. The first type is highly malleable tasks, with their sequential ratios randomly distributed between 0.05 and 0.15. The second type is poorly malleable tasks, with sequential ratios distributed between 0.85 and 0.95. The third type consists of hybrid tasks, with sequential ratios between 0.05 and 0.95. The numbers of processors are 4, 8, 16, 32 or 64. 10 random datasets are produced for each parameter combination. Therefore, there are totally 150 datasets for each parameter combination, and figures in Table 2 are the averages.

It can be noted that for most algorithms, the effectiveness ratios reduces as the task/processor ratio increases. The results produced by LPT are good when the task/processor ratio is larger than 3, which is consistent with our discussion in Section 3.2. Therefore, it can be concluded that LPT is the most practical algorithm when the number of tasks is much greater than the number of processors, because it is the simplest and fastest algorithm. It is clear that the improvement of OCM is great especially when the task/processor ratio is low. To conclude, the OCM is the most effective algorithm when the task/processor ratio is low, while LPT is practical when the ratio is high.

Table 3
Effectiveness ratios (%) with different numbers of processors.

#PROCESSOR	LPT	BA90	TWY92	LT94	MRT99	OCM
4	13.92	11.24	19.02	15.35	21.75	9.51
8	16.69	10.21	23.42	13.87	21.78	8.37
16	17.45	9.95	27.28	13.87	21.14	7.90
32	16.88	9.33	29.55	14.17	20.77	7.36
64	16.46	8.25	30.62	12.13	20.54	6.68

5.4. Effectiveness with different numbers of processors

Effectiveness ratios with various numbers of processors are displayed in Table 3. Similar to the previous sections, for each selected number of processors, 3 types of datasets (highly malleable, poorly malleable and hybrid) with task/processor ratios ranging from 1 to 3 are used, and 10 datasets for each parameter combination. So there are 90 random datasets for each selected number of processors, and the figures in Table 3 are the averages. From Table 3 it can be seen that the most effective algorithm is OCM again.

5.5. Hit rates of OCM

As discussed in Section 4.2, the OB algorithm needs to call the OM algorithm multiple times to get values of $C_{\max}^{\mathrm{OM}}(k)(k = 1, 2 \ldots n)$. Some of the calls may be eliminated according to Rule 1 of Section 4.5. If the value of $C_{\max}^{\mathrm{OM}}(k)$ is determined by another call to OM, and hence the call to OM for value k is skipped, we say that the call for value k is missed; otherwise, we say the call for value k is hit. Fig. 5 shows the hit rates for different values of k. The number of tasks is 100 for all the datasets, and the numbers of processors are 4, 8, 16, 32 and 64. Each parameter combination corresponds to 20 random datasets. So the results of Fig. 5 are the averages obtained from 100 datasets.

The hit rate for value 100 is 100%, because this is the first call to OM and cannot be skipped. The hit rate for 1 is also 100% because Rule 1 can hardly be applied to predict the value of $C_{\max}^{\mathrm{OM}}(1)$. Generally, the hit rate decreases as the value of k increases. This is advantageous since calling OM with larger values would require longer times. For all values greater than 32, the hit rates are not greater than 20%, so there are great chances of skipping these calls to OM.

From simulations on random datasets, we can draw the following conclusion. OCM is more effective than other existing algorithms for MPTS, especially for highly malleable tasks, and when the number of tasks is not too much larger than the number of processors. For cases where the number of tasks is far greater than the number of processors, or when tasks are poorly malleable, LPT is also a good choice, because it is simple and efficient.

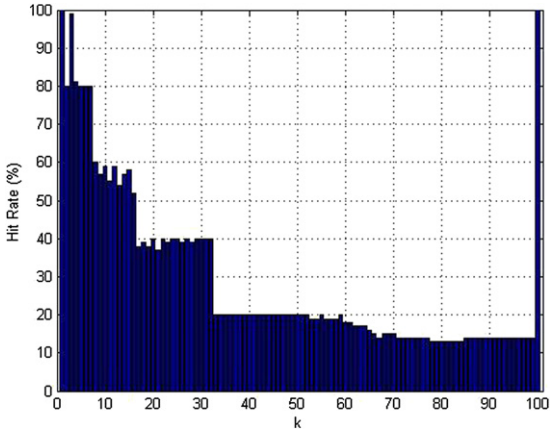

Fig. 5. Hit rates with different values.

5.6. Simulations on the standard benchmark suite

To estimate the effectiveness of different scheduling algorithms in real applications, we adopt the SPLASH-2 (2nd release of the Stanford Parallel Applications for Shared-Memory programs) benchmarks suite [41], which is one of the most popular benchmarks suites for parallel applications. It contains some of the most representative parallel applications, some of which are frequently encountered in practice. All the 14 applications of SPLASH-2 are listed in Table 4, and their detailed description can be found in [41].

The parallel computer used in our simulations has 2 Quad-Core AMD Opteron processors (each has 4 CPU cores), 64 GB memory, and Linux system. Each application in Table 4 corresponds to one task in our simulations. Applications of SPLASH-2 are implemented by some multi-threading parallel library, so they can be run on multiple processors in parallel and considered as malleable tasks in our simulations. The allotments of these malleable tasks are determined through a configuration file. The authors of SPLASH-2 recommended some default arguments for each application [41]. However, we find many applications violate the increasing work and decreasing time assumptions when default arguments are used. We believe the major reason is that the amounts of work associated with default arguments are too small for our parallel computer. For example, for 11 of the 14 applications, the sequential processing times are less than 1 s with default arguments (see Table 4). In practice, the sequential processing times of real applications are generally much longer. Due to these observations, we adjust arguments of these applications. The purpose is to increase the amounts of computation as slightly as possible, and

Table 4
Arguments for SPLASH-2.

APPLICATION NAME	ARGUMENTS		SEQUENTIAL TIME(S)	
	Default	Adjusted	Default	Adjusted
Barnes	16384 particles	32768 particles	6.846	13.795
Cholesky	Tk15.O	Tk29.O	0.285	0.717
Contiguous_blocks_lu	512 × 512 matrix	2048 × 2048 matrix	0.522	18.676
Contiguous_partitions _ocean	258 × 258 ocean	2050 × 2050 ocean	0.366	14.494
Fft	65536 points	65536 × 256 points	0.001	7.120
Fmm	16384 particles	65536 particles	0.175	4.923
Non_contiguous_ blocks_lu	512 × 512 matrix	2048 × 2048 matrix	0.008	24.215
Non_contiguous_ partitions_ocean	Time step 28800 s	Time Step 288 s	0.398	24.565
Radiosity	BFepsilon 0.015, room	BFepsilon 0.005, largeroom	7.819	16.637
Radix	1048576 integers	67108864 integers	0.018	5.257
Raytrace	Car	Ball4	0.663	6.904
Volrend	Head	No adjustment	1.120	–
Water-nsquared	Cutoff radius = 6.212752″	Cutoff radius = 62.12752″	0.087	1.163
Water-spatial	NPRINT = 3	NPRINT = 1	0.086	0.791

Table 5
Results of different algorithms on SPLASH-2.

ALGORITHM	4 CPU CORES		8 CPU CORES	
	ER (%)	Time (ms)	ER (%)	Time (ms)
LPT	3.689	6.083	39.995	6.187
Ba90	3.689	6.876	11.132	9.245
TWY92	5.359	8.440	13.793	15.508
LT94	3.689	6.388	38.833	6.616
MRT99	3.689	8.559	30.770	10.055
OCM	3.631	7.226	10.907	9.953

make the applications comply with the increasing work and decreasing time assumptions. Specifically, the default value of each argument is applied first, and then increased step by step, until the two assumptions hold. Table 4 gives the default and adjusted arguments, as well as their corresponding sequential processing times (in second).

Table 5 shows effectiveness ratios and execution times (in millisecond) of various algorithms on applications of SPLASH-2. The numbers of processors are 4 and 8, so the task/processor ratios are 3.5 and 1.75, respectively. The effective ratios produced by OCM are the smallest, indicating that OCM is more effective than other scheduling algorithms for real applications. The fastest algorithm is LPT, because it is the simplest. For 4 CPU cores, the effective ratio for OCM is 0.058% smaller than that of LPT, which means the makespan produced by OCM is $0.058\%C_{\mathrm{LB}}$ shorter than that produced by LPT, but OCM takes 1.143 ms longer than LPT. So if we evaluate an algorithm by the sum of the time cost by it and the makespan produced by it (that is, the total time for parallelizing all SPLASH-2 applications), we can conclude that if $C_{\mathrm{LB}} > 1.971$ (s), the overall performance of OCM is better, otherwise, LPT is better (Please refer to Section 5.1 for the definition of C_{LB}). As far as this dataset is concerned, we have

$$1.971 \ll 35.094 = \frac{1}{m}\sum_{j=1}^{n} t_j(1) \leq \frac{1}{m}\sum_{j=1}^{n} p_j^b t_j(p_j^b) = C_{\mathrm{LB}}.$$

In other words, the overall performance of OCM is better than that of LPT. Similarly, on 8 CPU cores, the effective ratio of OCM is 29.088% smaller, but it takes 3.770 ms longer, compared with LPT. For this dataset, the overall performance of OCM is better than LPT when $C_{\mathrm{LB}} > 0.013$ (s), and this holds similarly, since $0.013 \ll 17.547 = 1/m\Sigma t_j(1) \leq C_{\mathrm{LB}}$. In fact, it can be verified that for both 4 and 8 CPU cores, the overall performance of OCM is the best of all algorithms in Table 5.

6. Summary and future work

This paper deals with a frequently encountered NP-hard problem, the Malleable Parallel Task Scheduling problem (MPTS). MPTS is an extension of one of the most classic scheduling problems $P||C_{\max}$. For MPTS, a task is allowed to be processed simultaneously by more than one processor. One the one hand, such flexibility makes it possible to construct schedules with much smaller makespans, so more and more research has been devoted to MPTS in recent years. On the other hand, such flexibility dramatically increases the computing complexity of MPTS.

In this paper, some existing algorithms for MPTS are described with their features analyzed in detail. Based on these features and corresponding limits of these algorithms, we introduce our novel optimization algorithms and improving techniques and give detailed analyses to them. Combining these optimization algorithms and improving techniques gives birth to our 2-approximation algorithm for MPTS, we name it OCM (Optimizations Combined for MPTS). Extensive simulations on both random datasets and standard benchmark suite reveal that OCM is more effective than other existing algorithms, especially for tasks with high speedups, or when the number of tasks is not too much greater than the number of processors.

In the next step, we plan to further investigate the OCM algorithm in theory, and try to reduce its approximation ratio (which is 2 currently). Besides, we are going to conduct more extensive simulations to examine the performance of OCM in a wider range of situations.

References

[1] G.M. Amdahl, The validity of the single processor approach to achieving large-scale computing capabilities. Proceedings of AFIPS Spring Joint Computer Conference, AFIPS Press, pp. 483–485, 1967.
[2] A.K. Amoura, E. Bampis, C. Kenyon, Y. Manoussakis, Scheduling independent multiprocessor tasks, Algorithmica 32 (2) (2008) 247–261.
[3] B. Baker, E. Coffman, R. Rivest, Orthogonal packings in two dimensions, SIAM J. Comput. 9 (4) (1980) 846–855.
[4] K.B.P. Banerjee, Approximate algorithms for the partitionable independent task scheduling problem, in: Proceedings of the 1990 International Conference on Parallel Processing, vol. I, 1990, pp. 72–75.
[5] J. Blazewicz, M. Drabowski, J. Weglarz, Scheduling multiprocessor tasks to minimize schedule length, IEEE Trans. Comput. 35 (5) (1986) 389–393.
[6] J. Blazewicz, M. Machowiak, G. Mounié, D. Trystram, Approximation algorithms for scheduling independent malleable tasks, Proceedings of the. Seventh European Conference on. Parallel Computing, Lecture Notes in Computer Science, vol. 2150, pp. 191–197, 2001.
[7] J. Blazewicz, M. Machowiak, J. Weglarz, M.Y. Kovalyov, D. Trystram, Scheduling malleable tasks on parallel processors to minimize the makespan, Ann. Oper. Res. 129 (1-4) (2004) 65–80.
[8] M. Bougeret, P-F. Dutot, K. Jansen, C. Otte, D. Trystram, A fast 5/2-approximation algorithm for hierarchical scheduling. in: Proceedings of the 16th International Euro-Par Conference, Part I, pp. 157–167, 2010.
[9] J. Chen, C.Y. Lee, General multiprocessor task scheduling, Nav. Res. Logist. 46 (1) (1999) 57–74.
[10] E.G. Coffman Jr., M.R. Garey, D.S. Johnson, R.E. Tarjan, Performance bounds for level-oriented two-dimensional packing algorithms, SIAM J. Comput. 9 (4) (1980) 801–826.
[11] K. Efe, V. Krishnamoorthy, Optimal scheduling of compute-intensive tasks on a network of workstations, IEEE Trans. Parallel Distrib. Syst. 6 (6) (1995) 668–673.
[12] A. Feldmann, J. Sgall, S.H. Teng, Dynamic scheduling on parallel machines, Lecture Notes in Comput. Sci. 130 (1) (1994) 49–72.
[13] M. Garey, R. Graham, Bounds for multiprocessor scheduling with resource constraints, SIAM J. Comput. 4 (2) (1975) 187–200.
[14] R.L. Graham, Bounds for certain multiprocessing anomalies, Bell Syst. Tech. J. 45 (9) (1966) 1563–1581.
[15] R.L. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Appl. Math. 17 (2) (1969) 263–269.
[16] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Ann. Discrete Math. 5 (1979) 287–326.
[17] L.A. Hall, Approximation Algorithms for Scheduling, Approximation algorithms for NP-hard problems, PWS Publishing Corporation, 1996, pp. 1–45.
[18] J.T. Havill, W. Mao, Competitive online scheduling of perfectly malleable jobs with setup times, European J. Oper. Res. 187 (2008) 1126–1142.
[19] D.S. Hochbaum, D.B. Shmoys, Using dual approximation algorithms for scheduling problems: theoretical and practical results, J. ACM 34 (1) (1987) 144–162.
[20] K. Jansen, Scheduling malleable parallel tasks: an asymptotic fully polynomial time approximation scheme, Algorithmica 39 (1) (2004) 59–81.
[21] K. Jansen, L. Porkolab, Linear-time approximation schemes for scheduling malleable parallel tasks, Algorithmica 32 (3) (2002) 507–520.
[22] K. Jansen, L. Porkolab, Computing optimal preemptive schedules for parallel tasks: linear programming approaches, Math. Program. 95 (3) (2003) 617–630.
[23] K. Jansen, R. Thöle, Approximation Algorithms for Scheduling Parallel Jobs: Breaking the Approximation Ratio of 2. Proceedings of the 35th international colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 5125, pp. 234–245, 2008.
[24] C. Kenyon, E. Remila, A near-optimal solution to a two-dimensional cutting stock problem, Math. Oper. Res. 25 (4) (2000) 645–656.
[25] J. Kleinberg, E. Tardos, Algorithm Design, Pearson Addison Wesley, 2006, 266-272.
[26] M.Y. Kovalyov, M. Machowiak, D. Trystram, J. Weglarz, Preemptable malleable task scheduling problem, IEEE Trans. Comput. 55 (4) (2006) 486–490.
[27] R. Krishnamurti, E. Ma, An approximation algorithm for scheduling tasks on varying partition sizes in partitionable multiprocessor systems, IEEE Trans. Comput. 41 (12) (1992) 1572–1579.
[28] J.Y.T. Leung, Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapman and Hall/CRC, 2004, 25-5–25-18.
[29] A. Lodi, S. Martello, M. Monaci, Two-dimensional packing problems: a survey, European J. Oper. Res. 141 (2) (2002) 241–252.
[30] W. Ludwig, P. Tiwari, Scheduling malleable and nonmalleable parallel tasks, in: Proceedings of the 5th ACM–SIAM Symposium on Discrete Algorithms, pp. 167–176. 1994.
[31] G. Mounié, C. Rapine, D. Trystram, Efficient approximation algorithms for scheduling malleable tasks, in: Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 23–32, 1999.

[32] G. Mounié, C. Rapine, D. Trystram, A 3/2-dual approximation algorithm for scheduling independent monotonic malleable tasks, SIAM J. Comput. 37 (2) (2007) 401–412.
[33] G.N.S. Prasanna, B.R. Musicus, Generalized multiprocessor scheduling using optimal control, in: Proceedings of Third Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 216–228, 1991.
[34] D. Sleator, A 2.5 times optimal algorithm for packing in two dimensions, Inform. Process. Lett. 10 (1) (1980) 37–40.
[35] A. Steinberg, A strip-packing algorithm with absolute performance bound 2, SIAM J. Comput. 26 (2) (1997) 401–409.
[36] J. Turek, J. Wolf, P. Yu, Approximate algorithms for scheduling parallelizable tasks. in: Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 323–332, 1992.
[37] Q. Wang, K.H. Cheng, List scheduling of parallel tasks, Inform. Process. Lett. 37 (5) (1991) 291–297.
[38] Q. Wang, K.H. Cheng, A heuristic of scheduling parallel tasks and its analysis, SIAM J. Comput. 21 (2) (1992) 281–294.
[39] J. Weglarz, Project scheduling with continuously-divisible, doubly constrained resources, Manage. Sci. 27 (9) (1981) 1040–1052.
[40] J. Weglarz, Modelling and control of dynamic resource allocation project scheduling systems, in: S.G. Tzafestas (Ed.), Optimization and Control of Dynamic Operational Research Models, North-Holland, Amsterdam, 1982, pp. 105–140.
[41] S.C. Woo, M. Ohara, E. Torrie, et al. The SPLASH-2 Characterization and Methodological Considerations, in: Proceedings of the 22nd Annual Int. Symp.Computer Architecture, Santa Margherita, pp. 24–36, 1995.

Fan Liya was born in 1982. He has a Ph.D. from the Institute of Computing Technology, Chinese Academy of Sciences, and is a staff researcher of IBM China Research Lab. His research interests include parallel computing and scheduling algorithms.

Zhang Fa was born in 1974. He is an associate professor of Institute of Computing Technology, Chinese Academy of Sciences. His research interests include high performance algorithms and bioinformatics.

Wang Gongming was born in 1981. He is a Ph.D. candidate of Institute of Computing Technology, Chinese Academy of Sciences, and a member of China Computer Federation. His research interests include computer graphics, digital image processing and visualization.

Liu Zhiyong was born in 1946. He is a professor of Institute of Computing Technology, Chinese Academy of Sciences, and a senior member of China Computer Federation. His research interests include high performance algorithm and architecture, and parallel processing.