Scheduling Parallel Identical Machines to Minimize Makespan: A Parallel Approximation Algorithm

Laleh Ghalami[1], Daniel Grosu[1,*]

[a]*Department of Computer Science, Wayne State University, 5057 Woodward Avenue, Detroit, MI 48202, USA*

Abstract

Approximation algorithms for scheduling parallel machines have been studied for decades, leading to significant progress in terms of their approximation guarantees. The algorithms that provide near optimal performance are not feasible to use in practice due to their huge execution time requirements, thus underscoring the importance of developing efficient parallel approximation algorithms with near-optimal performance guarantees that are suitable for execution on current parallel systems, such as multi-core systems. We present the design and analysis of a parallel approximation algorithm for the problem of scheduling jobs on parallel identical machines to minimize makespan. The design of the parallel approximation algorithm is based on the best existing polynomial-time approximation scheme (PTAS) for the problem. To the best of our knowledge, this is the first practical parallel approximation algorithm for the minimum makespan scheduling problem that maintains the approximation guarantees of the sequential PTAS and it is specifically designed for execution on shared-memory parallel machines. We implement and run the algorithm on a large multi-core system and perform an extensive experimental analysis on data generated from realistic probability distributions. The results show that our proposed parallel approximation algorithm achieves significant speedup with respect to both the sequential PTAS and the CPLEX-based solver that solves the mixed integer program formulation of the problem.

Keywords: Scheduling, Approximation algorithms, Parallel algorithms.

*Corresponding author

Email addresses: laleh.ghalami@wayne.edu (Laleh Ghalami), dgrosu@wayne.edu (Daniel Grosu)

1. Introduction

Approximation algorithms for various combinatorial optimization problems have been studied for decades, leading to significant progress in terms of their approximation guarantees [1]. Despite the huge advances in parallel computing technologies over the past few decades, parallel approximation algorithms have been developed for only a small number of combinatorial optimization problems such as vertex cover [2], set cover [3, 4], facility location [5] and k-center [6]. The design of these parallel approximation algorithms focused on obtaining polylogarithmic time (i.e., $O(\log^c n)$) on PRAMs (Parallel Random Access Machines). Despite the fact that the problem of scheduling parallel identical machines is one of the most studied combinatorial optimization problems, there is only one published paper that investigated the design of a PRAM approximation algorithm for solving it [7]. The existing sequential approximation algorithm that provides near-optimal solutions to the problem (proposed by Hochbaum and Shmoys [8]) is not feasible to use in practice due to its huge execution time requirements, thus underscoring the importance of developing efficient parallel approximation algorithms with near-optimal performance guarantees that are suitable for execution on current parallel systems, such as multi-core systems. Therefore, in this paper we focus on designing a parallel approximation algorithm for the problem of scheduling parallel identical machines.

Using the notation for scheduling problems proposed by Lawler et al. [9], the problem of scheduling jobs on parallel identical machines is denoted by $P \mid\mid C_{max}$, where P specifies that the environment is composed of parallel identical machines, and C_{max} specifies the objective, that is, the minimization of the makespan. The $P \mid\mid C_{max}$ problem is defined as follows:

> $P \mid\mid C_{max}$ *Problem*: We are given a set of n jobs that need to be scheduled on m identical machines (running in parallel). Each job j, $j = 1, \ldots, n$, requires t_j units of time for processing, becomes available for processing at time zero, and once assigned to a processor for execution cannot be preempted. Each machine cannot process more than one job at a time. The objective is to *minimize the makespan* C_{max}, where $C_{max} := \max_{j=1,\ldots,n} C_j$, and C_j is the completion time of

job j, $j = 1, \ldots, n$. In other words, the objective is to minimize the maximum completion time of the jobs.

The problem $P \mid\mid C_{max}$ is NP-hard [10], therefore, unless P = NP, there does not exist a polynomial time algorithm that finds an optimal solution for it. Thus, we have to rely on approximation algorithms for solving it. A *c-approximation algorithm* for an optimization problem is a polynomial time algorithm that, for all instances of the problem, finds a solution whose value is within a factor of c from that of the optimal solution. We call c, the *approximation ratio* of the algorithm. For maximization problems, $c < 1$, while for minimization problems, $c > 1$. For some optimization problems it is possible to obtain very good approximations, called *Polynomial-Time Approximation Schemes* (PTAS). A PTAS is a family of algorithms $\{A_\epsilon\}$ such that for each ϵ there is an algorithm A_ϵ that is a $(1+\epsilon)$-approximation algorithm (for minimization problems), or $(1-\epsilon)$-approximation algorithm (for maximization problems). The running time of a PTAS could be exponential on $1/\epsilon$ or even worse. A *Fully Polynomial-Time Approximation Scheme* (FPTAS) requires that the algorithms A_ϵ are polynomial on both the problem size and $1/\epsilon$ [1].

Several approximation algorithms, with different approximation guarantees, have been proposed for $P \mid\mid C_{max}$. One of the simplest is the *list scheduling* (LS) algorithm. The algorithm assigns a job from an arbitrarily ordered list whenever a machine becomes available. It guarantees an approximation ratio of 2. The approximation guarantee of LS can be improved to 4/3 by sorting the jobs in non-increasing order of their processing times and then apply LS on the ordered list. This improved version of LS is called the *longest processing time* (LPT) algorithm. Since $P \mid\mid C_{max}$ is strongly NP-hard [11], a PTAS provides the strongest approximation guarantee one could obtain, unless P=NP. That is, no FPTAS exists for $P \mid\mid C_{max}$, unless P=NP. Such a PTAS was proposed by Hochbaum and Shmoys [8]. Since this algorithm provides the best approximation guarantee for the problem, in this paper, we focus on designing a parallel approximation algorithm based on it that provides the same approximation guarantees and is suitable for execution on multi-core systems.

1.1. Related Work

Since Karp [10] showed that the $P \mid\mid C_{max}$ problem is NP-hard, the majority of research has been directed towards deriving the approximation bounds of the sequential approximation algorithms for solving it. Graham [12] showed that LS is a 2-approximation algorithm. He later proved that LPT is a 4/3-approximation algorithm [13]. Coffman et al. [14] proposed the *multifit algorithm* (MF) which is based on techniques used in the bin-packing problem. The key idea behind MF is to find the smallest number of machines that can accommodate all jobs. The jobs are sorted in order of non-increasing processing times and each job is placed into the first machine which it will fit. If the number of attempts to find the smallest number of machines is k (by binary search), the makespan is at most $1.22 + 2^{-k}$. Friesen [15] improved the bound to $1.2 + 2^{-k}$. A tighter bound of $\frac{13}{11} + 2^{-k}$ has been obtained by Yue [16]. Friesen and Langston [17] developed an algorithm with the same running time as MF but with an improved bound of $\frac{72}{61} + 2^{-k}$. Sahni [18] proposed a FPTAS for the special case in which the number of parallel machines is fixed. If the number of machines is not fixed, no FPTAS exists, unless P = NP [11]. Hochbaum and Shmoys [8] used a variation of MF to develop a PTAS for $P \mid\mid C_{max}$, assuring that with a relative error of ϵ, it executes in time $O((n/\epsilon)^{1/\epsilon^2})$. This PTAS is the basis for the design of our proposed parallel approximation algorithm.

There is very limited research on the design of parallel algorithms for $P \mid\mid C_{max}$. Helmbold and Mayr [19] showed that the problem of obtaining LS schedules is P-complete. A problem is P-complete if it is a problem in P and every problem in P is reducible to it in polylogarithmic time on a PRAM with polynomial number of processors. P-complete problems are believed to be inherently sequential. Hence, it is unlikely to obtain an efficient parallel algorithm producing LS schedules. The closest work to ours is by Mayr [7] who designed a parallel $(1+\epsilon)$-approximation algorithm for $P \mid\mid C_{max}$ that runs in time $O(\log^2 n)$ on an EREW-PRAM with a polynomial number of processors. Their algorithm was designed for the theoretical PRAM model requiring a polynomial number of processors, which makes it unrealistic to use in practice. However, our proposed parallel approximation algorithm is designed for

multi-core systems and provides the same approximation guarantees as the PTAS.

1.2. Our Contributions

We address two issues that were not considered in the traditional design of approximation algorithms for the $P \mid\mid C_{max}$ problem, which mainly focused on developing sequential approximation algorithms with various approximation guarantees. That is, we take into account the huge computing power offered by the current parallel computing technologies and exploit the potential parallelism when designing approximation algorithms for this problem. We design a parallel approximation algorithm for $P \mid\mid C_{max}$ based on the PTAS proposed by Hochbaum and Shmoys [8]. The proposed algorithm provides the same approximation guarantees as the PTAS and is specifically designed for multi-core systems. To the best of our knowledge this is the first parallel approximation algorithm for solving the problem on shared-memory systems, proposed in the literature. We implement and run the algorithm on a multi-core system and perform an extensive experimental analysis. The results show that our proposed parallel approximation algorithm achieves significant speedup with respect to both the sequential PTAS and the CPLEX-based solver that solves the integer program formulation of the problem. This paper is a revised and extended version of [20].

1.3. Organization

The rest of the paper is organized as follows. In Section 2, we describe and analyze the sequential PTAS algorithm that is the basis for the design of our parallel approximation algorithm. In Section 3, we present our parallel approximation algorithm for the minimum makespan problem. In Section 4, we characterize the properties of our proposed parallel approximation algorithm. In Section 5, we analyze the performance of our algorithm by performing extensive experiments. In Section 6, we conclude the paper and present possible directions for future research.

2. A PTAS for $P \mid\mid C_{max}$

We present the PTAS for $P \mid\mid C_{max}$ proposed by Hochbaum and Shmoys [8], which is the basis for the design of our proposed parallel approximation algorithm. The basic idea of

their PTAS is to partition the set of jobs into two sets, long and short jobs, round down the processing times of the long jobs, and find an optimal schedule for the rounded long jobs. Then, consider the schedule of the rounded long jobs as the schedule of the long jobs with their original processing times, and finally, extend the schedule by scheduling the short jobs using LPT. The PTAS is given in Algorithm 1.

We now describe the PTAS in more details. The algorithm requires as input, the number of machines, m; the number of jobs, n; the processing times of the jobs t_j, $j = 1, \ldots, n$; and the relative error $\epsilon > 0$. We denote by $\mathcal{T}$ the multiset of jobs' processing times, i.e., $\mathcal{T} = \{t_1, ..., t_n\}$, and assume that all jobs' processing times are positive integers. The algorithm starts by computing the lower and upper bounds (denoted by LB and UB) on the optimal makespan of the set of n jobs on m identical machines (Lines 2-3). The upper and lower bounds are given by:

$$LB = \max\left\{\left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil, \max_{j=1,\ldots,n} t_j\right\} \tag{1}$$

and

$$UB = \left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil + \max_{j=1,\ldots,n} t_j. \tag{2}$$

It is easy to see that the optimal schedule is within $[LB, UB]$. LB is given by the maximum between the total processing time of the jobs divided by the number of machines, and the processing time of the longest job. Similarly, UB is given by the sum of the processing time of the longest job, and the total processing times of the jobs divided by the number of machines.

The algorithm performs a bisection search procedure for a target makespan value T on the interval $[LB, UB]$ and determines a schedule for the long jobs that fits within T (Lines 5-30). The multiset $\mathcal{T}$ of the processing times of the jobs is partitioned into two multisets: $\mathcal{L}$, containing the processing times of the long jobs, where a long job is a job j with $t_j > T/k$; and $\mathcal{S}$, containing the processing times of the short jobs, i.e., jobs with $t_j \leq T/k$, where $k = \lceil 1/\epsilon \rceil$ (Lines 9-13). Next, it rounds down the processing times of the long jobs to their nearest multiples of $\lfloor T/k^2 \rfloor$ and includes them in the multiset $\tilde{\mathcal{L}}$ of rounded processing times

Algorithm 1 PTAS for $P||C_{max}$ (Hochbaum and Shmoys [8])

1: **Input:** $n, m, \mathcal{T} = \{t_1, ..., t_n\}, \epsilon$
2: $LB \leftarrow \max\left\{\left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil, \max_{j=1,...,n} t_j\right\}$
3: $UB \leftarrow \left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil + \max_{j=1,...,n} t_j$
4: $k = \lceil 1/\epsilon \rceil$
5: **while** $LB < UB$ **do**
6: $\quad T = \lfloor (UB + LB)/2 \rfloor$
7: $\quad \mathcal{S} \leftarrow \emptyset$
8: $\quad \mathcal{L} \leftarrow \emptyset$
9: $\quad$ **for all** $t \in \mathcal{T}$ **do**
10: $\quad\quad$ **if** $t \leq T/k$ **then**
11: $\quad\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{t\}$
12: $\quad\quad$ **else**
13: $\quad\quad\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{t\}$
14: $\quad \tilde{\mathcal{L}} \leftarrow \emptyset$
15: $\quad$ **for all** $t \in \mathcal{L}$ **do**
16: $\quad\quad$ **for** $i = 1, ..., k^2$ **do**
17: $\quad\quad\quad$ **if** $i\lfloor \frac{T}{k^2} \rfloor \leq t < (i+1)\lfloor \frac{T}{k^2} \rfloor$ **then**
18: $\quad\quad\quad\quad \tilde{\mathcal{L}} \leftarrow \tilde{\mathcal{L}} \cup \{i\lfloor \frac{T}{k^2} \rfloor\}$
19: $\quad$ **for** $i = 1, ..., k^2$ **do**
20: $\quad\quad n_i = 0$
21: $\quad\quad$ **for all** $t \in \tilde{\mathcal{L}}$ **do**
22: $\quad\quad\quad$ **if** $t = i\lfloor T/k^2 \rfloor$ **then**
23: $\quad\quad\quad\quad n_i = n_i + 1$
24: $\quad N = (n_1, ..., n_{k^2})$
25: $\quad OPT = DP(N, T)$
26: $\quad$ Obtain schedule $\tilde{\mathbf{J}} = \{\tilde{\mathbf{J}}_1, \tilde{\mathbf{J}}_2, \ldots, \tilde{\mathbf{J}}_m\}$ from DP-table
27: $\quad$ **if** $OPT \leq m$ **then**
28: $\quad\quad UB = T$
29: $\quad$ **else**
30: $\quad\quad LB = T + 1$
31: **for** $i = 1, ..., m$ **do**
32: $\quad w_i = 0$
33: $\quad$ **for all** $\tilde{t} \in \tilde{\mathbf{J}}_i$ **do**
34: $\quad\quad$ **for all** $t \in \mathcal{L}$ **do**
35: $\quad\quad\quad$ **if** $\tilde{t} \leq t < \tilde{t} + \lfloor \frac{T}{k^2} \rfloor$ **then**
36: $\quad\quad\quad\quad \mathbf{J}_i \leftarrow \mathbf{J}_i \cup \{t\}$
37: $\quad\quad\quad\quad w_i = w_i + t$
38: $\quad\quad\quad\quad \mathcal{L} = \mathcal{L} \setminus \{t\}$
39: $\quad\quad\quad\quad$ **break**
40: $\quad \mathbf{J} \leftarrow \mathbf{J} \cup \{\mathbf{J}_i\}$
41: Sort jobs from $\mathcal{S}$ in non-increasing order of processing times
42: **for all** $t \in \mathcal{S}$ **do**
43: $\quad min \leftarrow \infty$
44: $\quad j = 1$
45: $\quad$ **for** $i = 1, ..., m$ **do**
46: $\quad\quad$ **if** $w_i < min$ **then**
47: $\quad\quad\quad min = w_i$
48: $\quad\quad\quad j = i$
49: $\quad \mathbf{J}_j \leftarrow \mathbf{J}_j \cup \{t\}$
50: $\quad w_j = w_j + t$
51: **return J**

of long jobs (Lines 15-18). Then, the algorithm determines the number of jobs of each of the rounded sizes and creates a k^2-dimensional vector $N = (n_1, \ldots, n_{k^2})$, where n_i is the number of long jobs of rounded size equal to $i\lfloor T/k^2 \rfloor$, $i = 1, \ldots, k^2$ (Lines 19-24). After creating

Algorithm 2 $DP(N,T)$

Input: $N = (n_1, ..., n_{k^2}), T$
$optimal \leftarrow \infty$
Generate the set $\mathcal{C}$ of all possible machine configurations $(s_1, ..., s_{k^2})$, where $\sum_{i=1}^{k^2} i \left\lfloor \frac{T}{k^2} \right\rfloor s_i \leq T$
for all $(s_1, ..., s_{k^2}) \in \mathcal{C}$ **do**
    $opt = DP(n_1 - s_1, ..., n_{k^2} - s_{k^2})$
    if $optimal > opt$ **then**
        $optimal = opt$
return $optimal + 1$

the vector N, the algorithm finds a schedule for the long rounded jobs with a makespan within time T. This is done by employing the dynamic programming algorithm DP, given in Algorithm 2. The dynamic programming algorithm determines the suitable number of machines to achieve a makespan within T and the schedule $\tilde{\mathbf{J}} = \{\tilde{\mathbf{J}}_1, \tilde{\mathbf{J}}_2, \ldots, \tilde{\mathbf{J}}_m\}$ of the long jobs of rounded sizes, where $\tilde{\mathbf{J}}_i$ is the set of long jobs assigned to machine i.

The DP algorithm generates the set $\mathcal{C}$ of all possible machine configurations. A *machine configuration* is a k^2-dimensional vector $(s_1, \ldots, s_{k^2})$ specifying an assignment of tasks to one machine, where s_i is the number of long jobs of rounded size equal to $i\lfloor T/k^2 \rfloor$ assigned to that machine, and satisfying

$$\sum_{i=1}^{k^2} i \left\lfloor \frac{T}{k^2} \right\rfloor s_i \leq T. \tag{3}$$

To characterize the recurrence equation implemented by the DP algorithm, we denote by $OPT(n_1, \ldots, n_{k^2})$ the minimum number of machines sufficient to schedule the set of jobs given by the vector N and leading to a makespan within T. This is computed using the following recursive equation:

$$\begin{aligned} &OPT(n_1, \ldots, n_{k^2}) = \\ &1 + \min_{(s_1,\ldots,s_{k^2}) \in \mathcal{C}} OPT(n_1 - s_1, \ldots, n_{k^2} - s_{k^2}). \end{aligned} \tag{4}$$

The idea behind this recurrence is that a schedule assigns some jobs to one machine and then assigns the rest of the jobs to as few machines as possible. Hence, the values of

$OPT(n_1, \ldots, n_{k^2})$ are the components of a dynamic programming table. If the DP algorithm is successful to predict the number of machines correctly, it tells us that the algorithm produced a schedule with a makespan within T for the long jobs with rounded sizes, which we denote by $\tilde{\mathbf{J}}$. Therefore, we can interpret it as a schedule for the long jobs with original processing times with a makespan at most $(1 + \frac{1}{k})T$, then update $UB \leftarrow T$; otherwise, update $LB \leftarrow T + 1$ because the algorithm indicates that no feasible schedule of length T exists, and $T+1$ is a valid lower bound. This bisection search procedure continues until $UB = LB$. The procedure terminates after a polynomial number of iterations because the difference between the initial upper and lower bounds defined by (2) and (1) is at most $\max_{j=1,\ldots,n} t_j$, and it is halved each iteration. After the bisection search procedure is completed, the algorithm finds the corresponding schedule $\mathbf{J}$ of the long jobs with the original processing times using the schedule $\tilde{\mathbf{J}}$ (Lines 31-40). The last step consists of assigning the short jobs to the obtained schedule $\mathbf{J}$ by using the list longest processing time algorithm, LPT (Lines 41-51). The original algorithm by Hochbaum and Shmoys [8] employs LS instead of LPT to schedule the short jobs. In this paper, we employ LPT, which improves the performance of the algorithm in practice without changing the approximability guarantees of the original algorithm.

Hochbaum and Shmoys [8] showed that the final schedule obtained by the algorithm has a makespan of at most $(1 + \frac{1}{k})T$, therefore the algorithm is a $(1 + \epsilon)$-approximation algorithm. To show this, we assume that we found a schedule for the long jobs with a makespan at most $(1 + \frac{1}{k})T$ and then assign the short jobs to the schedule. We then pick a short job l, and assign it using LPT to the next available machine. Since $\frac{1}{m}\sum_{j=1}^{n} t_j \leq T$, we obtain $\frac{1}{m}\sum_{j \neq l} t_j \leq T$. This implies that

$$t_l + \frac{1}{m}\sum_{j \neq l} t_j < \frac{T}{k} + T = \Big(1 + \frac{1}{k}\Big)T. \tag{5}$$

This shows that the makespan of the obtained schedule is at most $(1 + \frac{1}{k})T$.

Because a machine configuration is a k^2-dimensional vector and each entry in the vector is an integer less than or equal to n, there are $O(n^{k^2})$ possible machine configurations considered

by the algorithm. Thus, in the worst case, the running time of the algorithm is exponential in k^2, and because $k = \lceil 1/\epsilon \rceil$, it is exponential in $O(1/\epsilon^2)$. This means that the algorithm is not a FPTAS, but a PTAS. The reason for not being able to obtain a FPTAS is that $P \mid\mid C_{max}$ is strongly NP-hard [11], and therefore, there does not exist a FPTAS for solving it, unless P=NP.

3. Parallel Approximation Algorithm

In this section, we introduce our proposed parallel approximation algorithm for $P \mid\mid C_{max}$. Our proposed parallel algorithm is based on parallelizing the PTAS proposed by Hochbaum and Shmoys [8] which was presented in the previous section. Since the dynamic programming procedure is the most expensive component of the PTAS presented in the previous section in terms of running time, and it dictates the complexity of the algorithm, we focus our efforts on parallelizing the DP algorithm (Algorithm 2). A dynamic programming formulation that contains a single recursive term is called *monadic*, while one that contains multiple recursive terms is called *polyadic* [21]. The dependencies between subproblems in a dynamic programming formulation can be modelled as a directed graph. If the graph is acyclic, these dependencies can be organized into levels such that one subproblem depends on some of the subproblems at previous levels. If the subproblems at all levels depend only on the subproblems at the previous level then the dynamic programming formulation is *serial*, otherwise it is *non-serial*. The dynamic programming formulation (Equation 4) of the PTAS presented in the previous section is *non-serial monadic*, which means that there is a single recursive term in the dynamic programming formulation and the subproblems at each level depend on subproblems at more than one previous level. These subproblems correspond to the components of a table which we call the *DP-table*.

The DP algorithm determines the optimal number of identical parallel machines needed for scheduling $\sum_{i=1}^{k^2} n_i$ jobs having a makespan within T, by computing the value of $OPT(n_1, \ldots, n_{k^2})$, where $(n_1, \ldots, n_{k^2})$ is a k^2-dimensional vector with the i-th component representing the number of long jobs of rounded size equal to $i\lfloor T/k^2 \rfloor$, $i = 1, \ldots, k^2$. Therefore, there are $\prod_{i=1}^{k^2} (n_i + 1)$ distinct entries in the DP-table and each entry takes at most $\lfloor 1/\epsilon \rfloor^{k^2}$ time to

compute. Since $\prod_{i=1}^{k^2}(n_i+1) = O(n^{k^2})$, the running time for filling out the entire table is $O((\frac{n}{\epsilon})^{(1/\epsilon)^2})$. This running time makes DP the most expensive component of the PTAS and motivates us to consider it for parallelization.

In order to obtain a parallel version of the DP algorithm we need to investigate the dependencies between the dynamic programming subproblems. Thus, we are interested to know how the DP algorithm (Algorithm 2) fills out the DP-table. To make the description easy to understand we consider a small example. We assume that we chose $\epsilon = 0.3$ in Algorithm 1, and thus, $k = 4$. Hence, the input vector N is a 16-dimensional vector. We also assume that we already obtained the target schedule time $T = 30$, and the following vector N in Algorithm 1:

$$N = (0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0). \tag{6}$$

This means that the instance of the problem considered in the example, has two long jobs of rounded size $6\lfloor T/k^2 \rfloor = 6$, and three long jobs of rounded size $11\lfloor T/k^2 \rfloor = 11$. Since there are only two non-zero entries in N, we will use $N = (2, 3)$ to denote the full vector. The DP algorithm starts with the input vector $N = (2, 3)$ and the goal is to obtain the value of $OPT(N)$. We assume that for $N = (2, 3)$, and the target makespan $T = 30$, the set of machine configurations $\mathcal{C}$, determined by the DP algorithm (Line 3), is as follows:

$$\begin{aligned} \mathcal{C} = \{ \quad & (0,0), (0,1), (0,2), (1,0), \\ & (1,1), (1,2), (2,0), (2,1) \}. \end{aligned} \tag{7}$$

Because, as in the case of vector N, the machine configuration vector has only two non-zero entries we denote it by a vector with two entries. For example, machine configuration $(1, 2)$ specifies that one long job of size 6 and two jobs of size 11 are assigned to a machine. Then, the algorithm uses the recursion from Equation (4) to calculate $OPT(N)$. It is clear from

Table 1: The Dynamic Programming Table For $N = (2, 3)$

	0	1	2	3
0	OPT(0,0)	OPT(0,1)	OPT(0,2)	OPT(0,3)
1	OPT(1,0)	OPT(1,1)	OPT(1,2)	OPT(1,3)
2	OPT(2,0)	OPT(2,1)	OPT(2,2)	OPT(2,3)

Equation (4) that we need the following subproblems to calculate $OPT(N)$:

$$\begin{aligned} OPT(2,3) = \; & 1 + \min\{OPT(2,3), OPT(2,2), \\ & OPT(2,1), OPT(1,3), OPT(1,2), \\ & OPT(1,1), OPT(0,3), OPT(0,2)\} \end{aligned} \tag{8}$$

The subproblems are obtained by subtracting from the vector $N = (2, 3)$ each of the machine configurations in $\mathcal{C}$. For example, $(1, 3)$ is obtained from $(2, 3) - (1, 0)$. Equation (8) can be simplified to

$$\begin{aligned} OPT(2,3) = \; & 1 + \min\{OPT(2,2), OPT(2,1), \\ & OPT(1,3), OPT(1,2), OPT(1,1), \\ & OPT(0,3), OPT(0,2)\}, \end{aligned} \tag{9}$$

by removing $OPT(2, 3)$, since it is obtained from the machine configuration $(0, 0)$, which means no assignment. The next step is calculating $OPT(2, 2)$ from Equation (9). If we continue this procedure, at the end we need to calculate $OPT(0, 0)$, which is 0. It is easy to check that these subproblems correspond to the entries in the DP table (Table 1). Therefore, in the sequential algorithm (Algorithm 2) the computation of the DP-table entries starts from the last entry, $(2, 3)$, and ends up at the first entry, $(0, 0)$.

Figure 1 shows the dependencies between subproblems, which clearly demonstrates that the DP formulation is non-serial monadic. We can easily determine from Figure 1 two important characteristics of the computation of subproblems. First, the flow of computation moves along the main diagonal, and second, the subproblems on each anti-diagonal (denoted by Level x, in Figure 1) are independent. Since the subproblems on an anti-diagonal are independent of each other, they can be processed in parallel, each by one processor. Fig-

ure 1 shows the assignment of the subproblems for the small example presented above to a parallel system composed of four processors. This is the approach we use in the design of our parallel algorithm. Similar approaches have been used to parallelize dynamic programming algorithms for other problems such as, the longest common subsequence and sequence alignment [22, 23].

The entries of the DP-table (Table 1) are $OPT(v)$, for all possible vectors $v = (v_1, v_2, \ldots, v_{k^2})$ with every v_j satisfying $v_j \leq n_j$, for $j = 1, \ldots, k^2$. We denote by V the one-dimensional array of vectors v obtained from the k^2-dimensional array of vectors v specifying the subproblems in the DP-table, by using the row-major order. The array V corresponding to vector $N = (2, 3)$ is:

$$\begin{aligned} V = \{ \quad & (0,0), (0,1), (0,2), (0,3), \\ & (1,0), (1,1), (1,2), (1,3), \\ & (2,0), (2,1), (2,2), (2,3)\}. \end{aligned} \tag{10}$$

In what follows, we show how the DP-table is filled out and how the value of $OPT(N)$ is computed. Since we consider the vector N as a 2-dimensional vector, then the DP-table has $(2 + 1)(3 + 1) = 12$ entries. We denote by σ, the number of entries in the DP-table, which also represents the size of the one-dimensional array V. In general, for a k^2-dimensional vector N, $\sigma = \prod_{i=1}^{k^2}(n_i + 1)$.

As mentioned above, the subproblems on an anti-diagonal of the DP-table are independent and the computation flow moves along the main diagonal. An important characteristic of the subproblems on a given anti-diagonal is that the sum of the components of the vector that specifies them is the same. This can be seen in Table 1. For instance, the sums of the elements of the vectors that specify subproblems $OPT(2, 0)$, $OPT(1, 1)$, and $OPT(0, 2)$ are the same, (i.e., 2). Since the subproblems on an anti-diagonal are independent they can be processed in parallel. The same sum of the components of the vectors indicates that the corresponding subproblems can be processed in parallel. The dependencies for subproblems

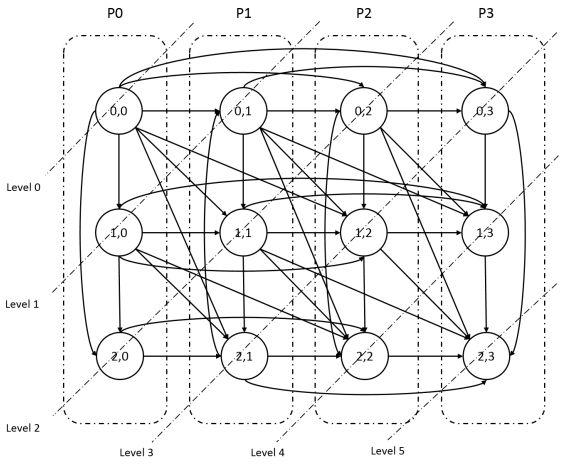


Figure 1: Dependency graph for $OPT(2,3)$

$OPT(2,0)$, $OPT(1,1)$ and $OPT(0,2)$ (marked with arrows) are as follows,

$$\begin{aligned} OPT(2,0) &\leftarrow \{OPT(1,0), OPT(0,0)\} \\ OPT(1,1) &\leftarrow \{OPT(1,0), OPT(0,1), OPT(0,0)\} \\ OPT(0,2) &\leftarrow \{OPT(0,1), OPT(0,0)\}. \end{aligned} \tag{11}$$

showing that they are all independent. Hence, they can be processed in parallel and assigned to different processors (cores). The idea of using the above sum to detect when the subproblems are independent applies to DP-tables with any number of dimensions.

We denote by d_i the sum of the components of the i-th vector $v^i = (v^i_1, \ldots, v^i_{k^2})$ in the array V, that is, $d_i = \sum_{j=1}^{k^2} v^i_j$, for $i = 0, \ldots, \sigma - 1$. We also denote by D the array $(d_0, \ldots, d_{\sigma-1})$. The order of the elements of the array D is the same as that of the elements of array V. For instance, if $v^7 = (1,3)$, then $d_7 = 1 + 3 = 4$. In the parallel algorithm the subproblems are assigned to processors according to their associated d_i values. The procedure starts from the subproblem $OPT(0,0)$, which has value 0. At each stage (on each anti-diagonal) the value of each subproblem in the preceding anti-diagonal is already obtained and the only task is to find the corresponding value according to its dependencies. For example, to calculate the value of $OPT(1,1)$, we need to know the values of $OPT(1,0)$, $OPT(0,1)$, and $OPT(0,0)$, which are all obtained in the previous (anti-diagonal) step (Figure 1). At

the end of the computation, the DP-table will be completely filled out and the DP function returns the optimal value, which is the value of the last element of the DP-table, that is, $OPT(N)$.

We now present the *Parallel DP* algorithm (Algorithm 3) in more details. The algorithm is designed for a shared-memory parallel system with P processors. The algorithm requires as input, the vector $N = (n_1, \ldots, n_{k^2})$, and the target makespan, T. The goal of the algorithm is to fill out the entire DP-table and find the value of the last subproblem in the table, that is, the optimal value of $OPT(N)$. It is worth to mention here that the recursive sequential version of the DP algorithm (Algorithm 2) starts from the last entry of the DP-table and recursively computes the other entries until it ends up at the first element. The parallel procedure of finding the optimal value of each subproblem can be handled in much the same way as in the recursive version, the only difference being that it starts from the first element of the array instead of from the last and is handled iteratively. Hence, the flow of computation sweeps along the main diagonal (Figure 1).

First, the algorithm determines the size of the DP-table, $\sigma = \prod_{i=1}^{k^2} (n_i + 1)$ (Line 2). Next, the P processors compute the sums of the distances of the vectors v^i, $i = 1, \ldots, \sigma$ in parallel (Lines 4-8). We express the parallelism using the "parallel for $i = 0, \ldots, \sigma - 1$ do" construct which specifies that each of the P processors will be assigned one iteration of the for loop in a round-robin fashion until all σ iterations are assigned. For this specific parallel for loop, each processor will be responsible for executing at most $\lceil \sigma/P \rceil$ iterations.

Since the total number of anti-diagonals of the DP-table is given by $n_1 + n_2 + \ldots + n_{k^2} + 1 = n' + 1$, the *Parallel DP* algorithm consists of $n' + 1$ sequential iterations, where n' is the number of long jobs. Each iteration l (corresponding to anti-diagonal l), $l = 0, \ldots, n'$, is executed by P processors in parallel (Lines 11-25). The subproblems on the same anti-diagonal are identified by checking their corresponding d_i values, that is, if the subproblems have the same d_i values, then they are independent and can be processed in parallel (Line 12). We denote by q_l the number of subproblems in each anti-diagonal. Note that the number of the subproblems assigned to each processor depends on q_l. If $q_l \geq P$ then each of the P

Algorithm 3 *Parallel* $DP(N,T)$

1: **Input:** $N=(n_1,\ldots,n_{k^2}),T$
2: $\sigma \leftarrow (n_1+1)(n_2+1)\ldots(n_{k^2}+1)$
3: Let $v^i=(v^i_1,\ldots,v^i_{k^2})$ and $OPT(v^i_1,\ldots,v^i_{k^2})$ be the i-th entry of DP-table in row-major order, where $i=0,\ldots,\sigma-1$
4: **parallel for** $i=0,\ldots,\sigma-1$ **do**
5: $\quad d_i=0$
6: $\quad$ **for** $j=0,\ldots,k^2-1$ **do**
7: $\quad\quad d_i=d_i+v^i_j$
8: **end parallel for**
9: $n'=n_1+\ldots+n_{k^2}$
10: **for** $l=0,\ldots,n'$ **do**
11: $\quad$ **parallel for** $i=0,\ldots,\sigma-1$ **do**
12: $\quad\quad$ **if** $d_i=l$ **then**
13: $\quad\quad\quad$ **if** $i=0$ **then**
14: $\quad\quad\quad\quad OPT(0,\ldots,0)\leftarrow 0$
15: $\quad\quad\quad\quad$ **break**
16: $\quad\quad\quad \mathcal{O}_{v^i}\leftarrow\emptyset$
17: $\quad\quad\quad \mathcal{C}_{v^i}\leftarrow$ all machine configurations of vector v^i
18: $\quad\quad\quad$ **for all** $(s_1,\ldots,s_{k^2})\in\mathcal{C}_{v^i}$ **do**
19: $\quad\quad\quad\quad \mathcal{O}_{v^i}\leftarrow\mathcal{O}_{v^i}\cup\{OPT(v^i_1-s_1,\ldots,v^i_{k^2}-s_{k^2})\}$
20: $\quad\quad\quad min\leftarrow\infty$
21: $\quad\quad\quad$ **for all** $o\in\mathcal{O}_{v^i}$ **do**
22: $\quad\quad\quad\quad$ **if** $min>o$ **then**
23: $\quad\quad\quad\quad\quad min=o$
24: $\quad\quad\quad OPT(v^i_1,\ldots,v^i_{k^2})\leftarrow min+1$
25: $\quad$ **end parallel for**
26: **return** $OPT(n_1,\ldots,n_{k^2})$

processors compute at most $\lceil q_l/P\rceil$ subproblems from anti-diagonal l; else q_l processors out of P, compute the q_l subproblems of anti-diagonal l, one per processor.

The operations corresponding to filling out the DP-table and finding the optimal value of each subproblem are executed in parallel by different processors for independent subproblems (Lines 11-25). In the following we describe these operations. First of all, one of the processors initializes the first subproblem, $OPT(0,\ldots,0)$ and sets its optimal value to 0. For computing the optimal value of a subproblem, we need to know its dependencies on the preceding subproblems and use them in Equation (4). Therefore, the algorithm generates the set $\mathcal{C}_{v^i}$ of all possible machine configurations, $(s_1,\ldots,s_{k^2})$, for vector v^i (Line 17). Note that $\mathcal{C}_{v^i}$

does not include the zero vector, since it means no assignment. Next, the algorithm reads the values of all subproblems $OPT(v_1^i - s_1, \ldots, v_{k^2}^i - s_{k^2})$ from the DP-table and places their optimal values into multiset $\mathcal{O}_{v^i}$ (Lines 18-19). Then, it determines the minimum among all values of the subproblems currently in $\mathcal{O}_{v^i}$, adds 1 to the minimum and assigns the value to subproblem $OPT(v_1^i, \ldots, v_{k^2}^i)$ (Lines 20-25). The ordering of iterations we establish using array D, guarantees that at each level the algorithm already computed all the needed preceding subproblems. At this point the parallel for loop terminates. Once all processors terminate all their tasks, the optimal value $OPT(N)$ is computed and returned (Line 26).

Our proposed parallel approximation algorithm for solving $P||C_{max}$ has the same structure as the sequential PTAS (Algorithm 1) with the exception that the sequential DP algorithm employed in Line 25 of Algorithm 1 is replaced by the *Parallel DP*. The other parts of the bisection search procedure (Lines 6-23 and 26-30, Algorithm 1), the procedure for replacing the rounded jobs with the jobs with original sizes (Lines 31-40, Algorithm 1), and the LPT procedure for allocation of short jobs (Lines 41-50) are kept the same as in the sequential PTAS since there is very little benefit that can be gained from parallelizing them. The reason is that the running time of these sequential parts is negligible when compared to the running time required to fill out the DP-table for reasonably sized instances of the problem.

4. Analysis of the parallel algorithm

We analyze the running time of the parallel approximation algorithm presented in Section 3 and prove the following result.

Proposition 1. *The running time of the proposed parallel approximation algorithm for scheduling parallel identical machines to minimize makespan is* $O((\frac{n}{\epsilon})^{(1/\epsilon)^2}/P)$.

Proof. We start by analyzing the running time of the main component of the algorithm, the *Parallel DP* algorithm. The runtime complexity of the *Parallel DP* algorithm depends on the amount of time required to process each subproblem, and the number of sequential iterations needed to finish all computations. The computational flow induced by the algorithm sweeps

along the main diagonal of a k^2-dimensional array. As in the previous section, we denote by q_l the number of entries in the DP-table that have the sum of the indices equal to l. Those entries correspond to anti-diagonal hyperplanes in the k^2-dimensional array (i.e., the DP-table). Since the total number of anti-diagonals is given by $n_1+n_2+\ldots+n_{k^2}+1 = n'+1$, the *Parallel DP* algorithm consists of $n'+1$ sequential iterations. Each iteration l (Lines 11-25), $l = 0, \ldots, n'$, is executed by P processors in parallel and involves the following:

(1) If $q_l \geq P$ then each of the P processors compute at most $\lceil q_l/P \rceil$ subproblems from diagonal l; else q_l processors out of P compute the q_l subproblems of diagonal l, one per processor.

(2) Each processor that has been assigned subproblems computes for each of those subproblems, a minimum among the values of subproblems from previous iterations as specified by the machine configurations, and adds 1 to the minimum.

After $n'+1$ iterations the DP-table is filled out and the $OPT(N)$ value is available in the last entry of the table. In each iteration, a processor computes the values of at most $\lceil q_l/P \rceil$ subproblems and since $q_l < n^{k^2-1}$ and each value can be computed in time $O(k^{k^2})$, then one iteration requires time $O(k^{k^2} n^{k^2-1}/P)$. Because *Parallel DP* performs $n^{'} + 1 = O(n)$ such iterations, the total time required to perform all iterations is $O(k^{k^2} n^{k^2}/P)$. Computing the elements of vector D (Lines 4-8) takes time $O(\sigma/P)$, and because $\sigma = O(n^{k^2})$, the time is $O(n^{k^2}/P)$. Therefore, the total running time of *Parallel DP* is $O(k^{k^2} n^{k^2}/P)$, that is $O((\frac{n}{\epsilon})^{(1/\epsilon)^2}/P)$. This is the dominant component of the running time of the proposed parallel approximation algorithm for $P \,||\, C_{max}$ and it represents its overall runtime complexity. □

5. Experimental Results

In this section, we investigate the performance of the proposed parallel approximation algorithm by performing extensive experiments on a multi-core system with 64 cores. We compare the performance of our proposed parallel algorithm in terms of running time, speedup and makespan, against that of optimal solution obtained by solving the mixed integer programming formulation of the problem, and that of two sequential approximation algorithms

PTAS and LPT. Since the running times of LPT are much smaller than the running time of the PTAS we will not include them in the plots showing the running times. We will only focus on comparing their performance in terms of makespan.

5.1. Experimental Setup

We perform extensive simulation experiments on several problem instances with different number of machines, number of jobs, and ranges for the distribution of the processing times of the jobs. We consider instances with $m = 20, 30, 40, 50$ machines and different ratios of the number of jobs and number of machines $\frac{n}{m} = 10, 20, 30$, and 40. Considering instances with different $\frac{n}{m}$ ratios allows us to investigate the influence of the size of the instances on the performance of the algorithms in a more structured way. Table 2 shows the sixteen types of instances (I1 to I16) considered in the experiments.

In order to investigate the performance of the algorithms on problem instances similar to those usually encountered in large scale parallel systems, we generate the processing times of the jobs using the Pareto distribution. Harchol-Balter and Downey [24] showed that the distribution of jobs processing times in parallel machines is Pareto (heavy-tailed). They studied different parallel systems and observed that the job processing times fit a curve t^{α}, with α varying from 0.8 to 1.3. We chose 0.99 for α and set the lower bound for the range of the processing times to 1, and the upper bound to 100, that is, $t_j \in [1, 100]$. For each type of instance, we generate 20 instances for a total of 320 instances and investigate the performance of our algorithm on those instances.

We use the CPLEX solver to solve the Mixed Integer Programming (MIP) formulation for $P \mid\mid C_{max}$ [25], which is as follows:

Table 2: Types of instances used in the experiments.

Instance	# of machines (m)	# of jobs (n)	$\frac{n}{m}$ ratio
I1	20	200	10
I2		400	20
I3		600	30
I4		800	40
I5	30	300	10
I6		600	20
I7		900	30
I8		1200	40
I9	40	400	10
I10		800	20
I11		1200	30
I12		1600	40
I13	50	500	10
I14		1000	20
I15		1500	30
I16		2000	40

$$
\begin{aligned}
&\mathsf{MIP:} \quad \min \quad y \\
&\text{s.t.} \\
&\qquad \sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1, \ldots, n, \\
&\qquad \sum_{j=1}^{n} t_j x_{ij} \leq y, \quad i = 1, \ldots, m. \\
&\qquad x_{ij} \in \{0, 1\} \qquad i = 1, \ldots, m; j = 1, \ldots, n
\end{aligned}
\tag{12}
$$

where x_{ij} is a binary variable which is 1 if job j is assigned to machine i, and 0 otherwise; and y is an integer variable whose optimal value is C_{max}. The first constraint ensures that each job is processed on exactly one machine. The second constraints ensure that y is at least as large as the total processing time on each machine. This formulation has $m + n$ constraints and $mn + 1$ variables.

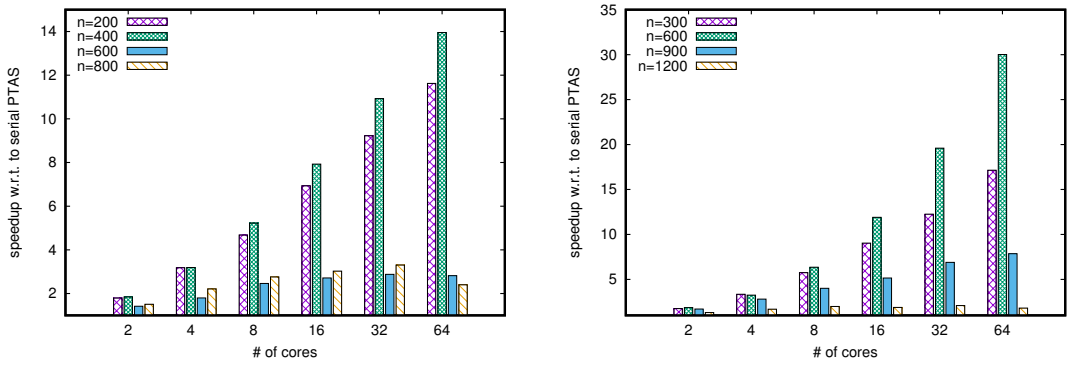
We implemented the parallel approximation algorithm in C++ and OpenMP. The paral-

lel approximation algorithm and the sequential PTAS are executed with $\epsilon = 0.3$. We choose this value of ϵ to obtain an approximation ratio for our algorithm that is below the approximation ratio of LPT. We also implemented two other sequential approximation algorithms, PTAS, and LPT in C++. We solve the mixed integer program formulation of the problem by using the CPLEX solver provided by IBM ILOG CPLEX Optimization Studio for Academics Initiative [26] and obtain the optimal solution. We denote by MIP (Mixed Integer Program) the CPLEX-based implementation that obtains the optimal solution. All programs implementing the algorithms were compiled using GCC version 4.8.5 and executed on a 64-core 2.4 GHz dual-processor AMD Opteron system with 512 GB of RAM and Red Hat Enterprise Linux Server as the operating system.

5.2. Analysis of Results

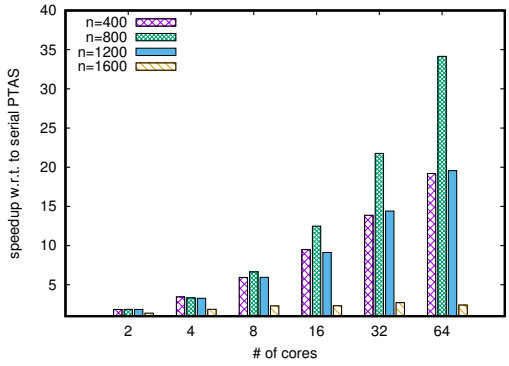
We first analyze the running time and the speedup obtained by the proposed parallel approximation algorithm. We consider two types of speedup for our analysis. The first type, the *speedup with respect to the sequential* PTAS, is defined as the ratio of the running time of the sequential PTAS algorithm and the running time of the parallel approximation algorithm. The second type, the *speedup with respect to* MIP is defined as the ratio of the running time of the CPLEX-based solver that solves the MIP and the running time of the parallel algorithm. We also present and analyze the actual approximation ratios with respect to the objective, which in our case is the makespan. Thus, the *actual approximation ratio* of an approximation algorithm for $P \mid\mid C_{max}$ is the ratio of the makespan obtained by the algorithm and the optimal makespan obtained by solving the MIP. We also analyze the effect of increasing the number of jobs, the number of machines, and the ratio between the number of jobs and number of machines on the speedup and the running times. We examine the effect of choosing different ϵ on the running times and the actual approximation ratio of the proposed parallel approximation algorithm.

Figure 2 shows the average speedup obtained by the proposed parallel approximation algorithm with respect to the sequential PTAS on a multi-core system with the number of cores ranging from 2 to 64 for all instances with different number of machines m, where

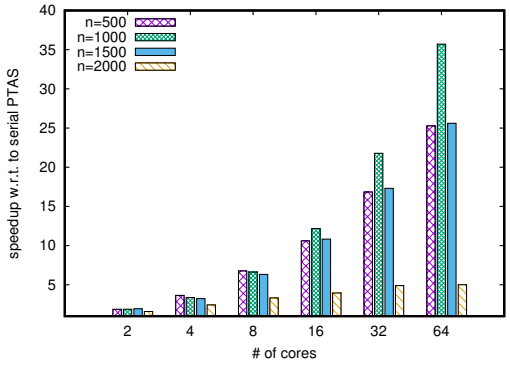


(a) Instances with m = 20.

(b) Instances with m = 30.



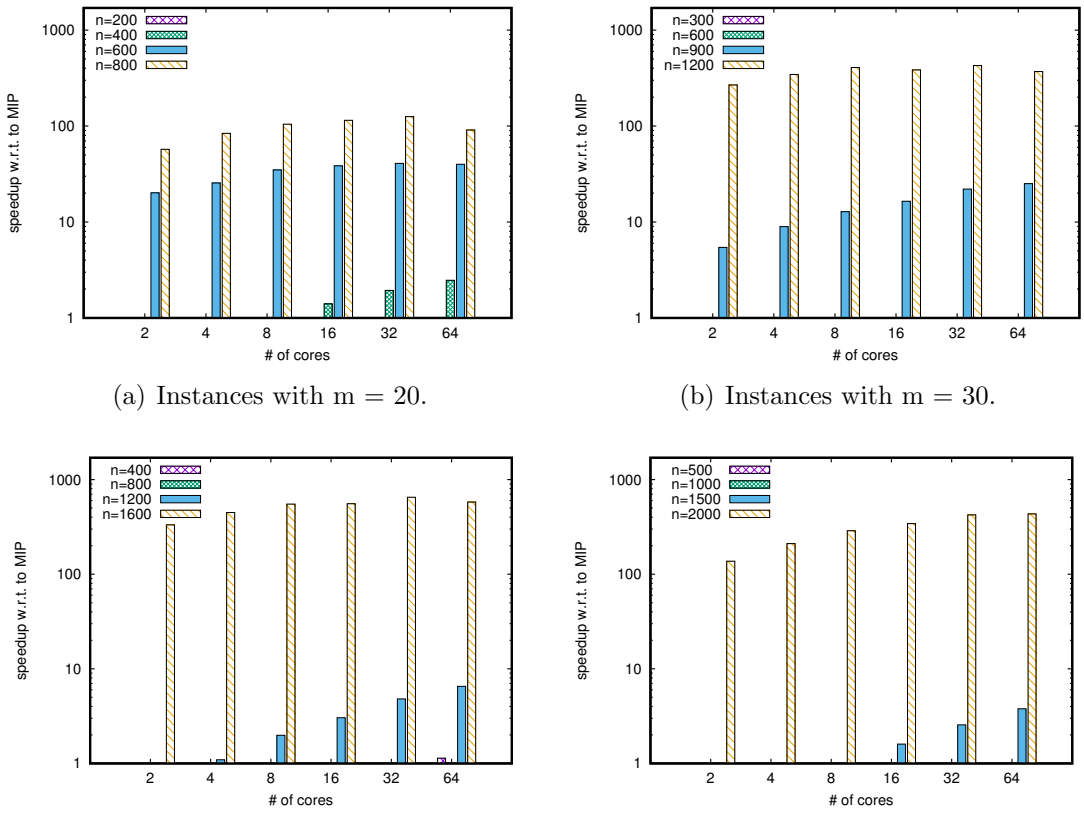
(c) Instances with m = 40.



(d) Instances with m = 50.

Figure 2: Average speedup with respect to the sequential PTAS vs. number of cores.

$m = 20, 30, 40, 50$ and different number of jobs n, ranging from 200 to 2000 depending on the type of instance. The values of the speedup for each type of instance are the averages over the speedup for 20 instances of that type. We observe that for instances with $\frac{n}{m} = 10$ and 20 as the number of machines and jobs increase, the speedup with respect to the sequential PTAS increases. The proposed parallel algorithm achieves very good speedup of up to 21.7 using 32 cores, and up to 35.7 using 64 cores for the instances with $m = 50$ and $n = 1000$. For instances with $\frac{n}{m} = 30$ and 40, the proposed parallel algorithm obtains smaller speedups (up to 4 for 16 cores and $m = 50$, $n = 2000$) than the one obtained for instances with $\frac{n}{m} = 10$ and 20. The reason for this is that the algorithm categorizes as small jobs the majority of the jobs (close to 99%) for instances with $\frac{n}{m} = 30$ and 40. Therefore, the small jobs are scheduled by using the LPT as the sequential algorithm does and the performance



(a) Instances with m = 20.

(b) Instances with m = 30.

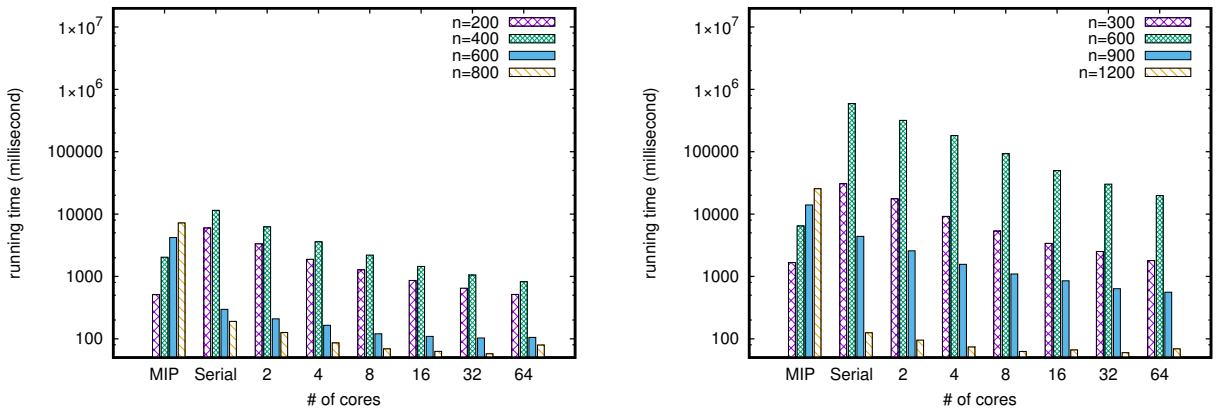
(c) Instances with m = 40.

(d) Instances with m = 50.

Figure 3: Average speedup with respect to MIP (CPLEX-based solver) vs. number of cores.

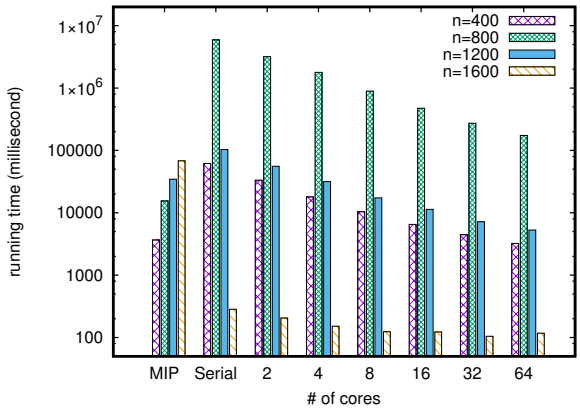
of the parallel algorithm is close to that of the sequential one, leading to small speedups. These instances represent in a sense the worst cases in terms of speedup for the proposed algorithm. These results show that the parallel algorithm is scalable for the instances with large number of large jobs and the size of the multi-core system used in the experiments. For those instances, we expect the speedup to increase as the number of cores increases past 64, but most likely the increases will not be as large as the ones obtained for the number of cores presented in the plots.

Figure 3 shows the average speedup with respect to the CPLEX-based solver that solves the MIP. Similar to the speedup with respect to the sequential PTAS, we plot the speedup for each type of instance and group them according to the number of machines. For small instances, our parallel algorithm requires a greater execution time than CPLEX, and there-

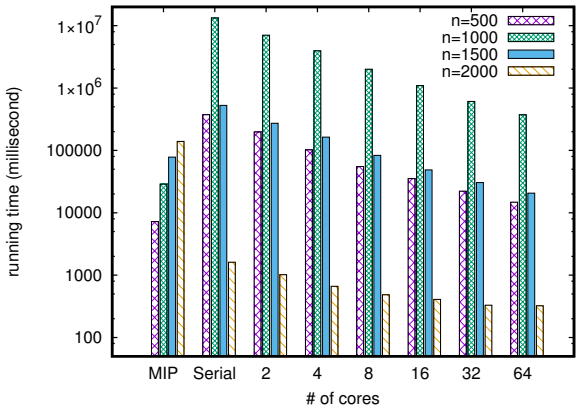


(a) Instances with m = 20.

(b) Instances with m = 30.



(c) Instances with m = 40.



(d) Instances with m = 50.

Figure 4: Average running time vs. number of cores.

fore, it does not achieve speedup. For those instances, there are no bars represented in the plots. For large instances with $\frac{n}{m} = 30$ and 40, our parallel algorithm obtains significant speedup for all the sizes of the multi-core system. For example, for instances with $\frac{n}{m} = 40$ and $n = 1600$ the speedup obtained on a system with 32 cores is 650. The increase in speedup for these types of instances is also significant from 332 for a system with two cores to 650 for a system with 32 cores. For small instances with $\frac{n}{m} = 10$ and 20 the obtained speedup is much smaller than the one obtained for the large instances. The maximum speedup achieved for these instances is about 2 on a multi-core system with 32 cores. This is due to the fact that CPLEX was able to find a solution in a smaller amount of time for these small instances compared to the other types of instances. We do not have enough details on the

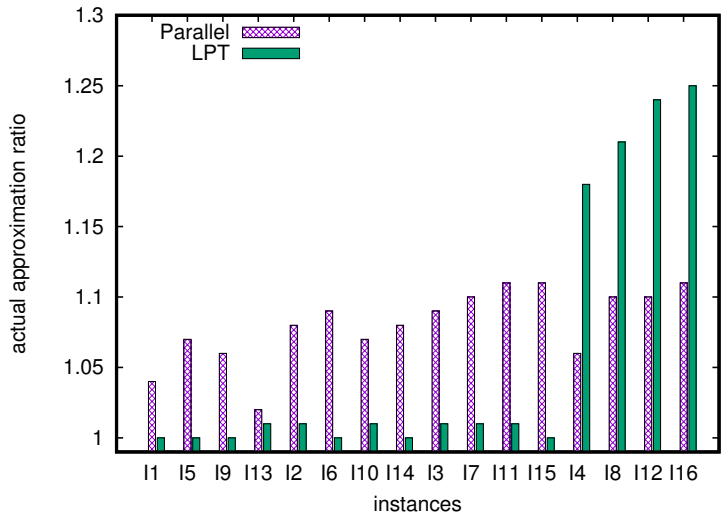


Figure 5: Actual approximation ratios

internal CPLEX implementation to be able to explain why this happened for those type of instances. In the case of large instances the parallel algorithm is able to scale well and obtain a significant speedup.

Figure 4 shows the average running times of the serial PTAS, the CPLEX-based solver (MIP) and the proposed parallel algorithm for all types of instances with different number of machines considered in the experiment. We observe that for instances with $\frac{n}{m} = 10$ and 20 (e.g., $m = 20$, $n = 200$ and 400) the running time of the parallel algorithm decreases significantly with the size of the multi-core system and is much smaller than the running time of the serial PTAS. For all large instances with $\frac{n}{m} = 40$ the proposed parallel algorithm obtains significantly smaller running times than the CPLEX-based solver. As an example for the instances with $m = 50$ and $n = 2000$ the CPLEX solver requires about 3 minutes while our parallel algorithm running on 64 cores requires only 322 milliseconds. For the instances with the number of jobs greater than 2000, the CPLEX-based solver is not even able to build the model of the problem and we are not able to perform the experiments with problem sizes larger than that. Our algorithm was still able to solve those instances in a reasonable amount of time.

The results presented and analyzed above focused on the running time and speedup.

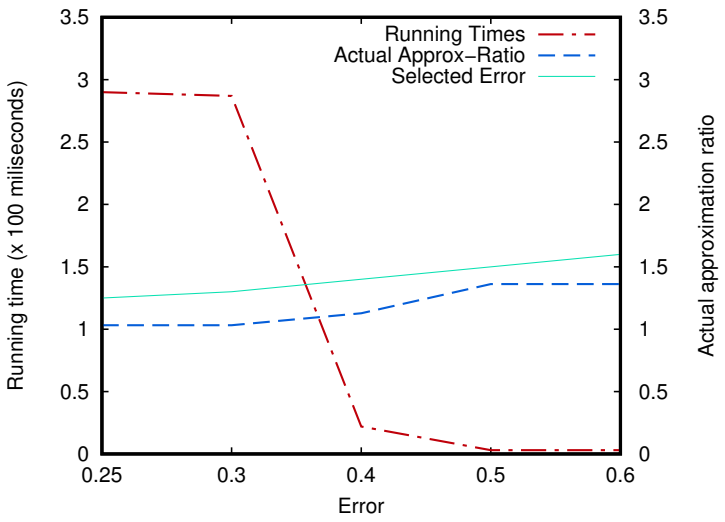


Figure 6: Actual approximation ratios and the running times vs. errors.

Next, we analyze the performance of the proposed algorithm in terms of actual approximation ratio. In order to do that we compare our algorithm against the LPT. In the following we will show that our algorithm performs better than LPT in terms of the actual approximation ratio for large instances. Figure 5 shows the comparison between the actual approximation ratio obtained by the proposed parallel algorithm and LPT for all instances. Similar to the running times and the speedups we take the average of the actual approximation ratios over all 20 samples of each type of instance. The figure shows that for instances with $\frac{n}{m} = 10$ (i.e., I1, I5, I9, and I13); $\frac{n}{m} = 20$ (i.e., I2, I6, I10, I14); and $\frac{n}{m} = 30$ (i.e., I3, I7, I11, I15), LPT performs better. However, as $\frac{n}{m}$ and the size of the problem increase the proposed parallel algorithm outperforms LPT. We can observe that for the instances with $\frac{n}{m} = 40$ (i.e., I4, I8, I12, and I16), the actual approximation ratio obtained by the proposed parallel algorithm is much smaller than that obtained by LPT. This is what we expected, our algorithm performs better in comparison to LPT for larger instances in terms of actual approximation ratio. Thus, the proposed parallel approximation algorithm is suitable for large instances of the problem and outperforms LPT for those instances.

We now investigate how the error factor ϵ affects the actual approximation ratio obtained by the proposed algorithm and the required running time for a medium size instance (i.e.,

I5, $n = 300$, $m = 30$). The actual approximation ratio is defined as the ratio of the makespan obtained by the corresponding algorithm (parallel approximation algorithm) and the makespan obtained by the CPLEX-based solution which produces the optimal schedule and makespan. Figure 6 shows the running times and the actual approximation ratio for different errors ranging from $\epsilon = 0.25$ to $\epsilon = 0.6$. As ϵ increases, we obtain a significant reduction in the running times (from 290 milliseconds to 3 milliseconds) while keeping the actual approximation ratio below the selected error. Therefore, by increasing the error we can reduce the running time and at the same time keep the actual approximation ratio much below the approximation guarantee provided by the algorithm (i.e., $1 + \epsilon$).

6. Conclusion

We proposed a parallel approximation algorithm for solving the problem of scheduling parallel identical machines to minimize makespan. The proposed algorithm provides the same approximation guarantees as the PTAS and was specifically designed for multi-core systems. To the best of our knowledge this is the first parallel approximation algorithm for solving the problem on shared-memory systems, proposed in the literature. We implemented the algorithm using OpenMP, performed extensive experiments on a large multi-core system, and analyzed its performance on data generated using realistic probability distributions. We compared the performance of our proposed parallel algorithm in terms of running time, speedup and makespan, against that of optimal solution obtained by solving the integer programming formulation of the problem, and that of another classical approximation algorithm LPT. The results showed that our proposed parallel approximation algorithm achieves significant speedup with respect to both the sequential PTAS and the CPLEX-based solver that solve the integer program formulation of the problem.

The results presented in this paper also show that despite the current held view that PTAS algorithms are of only pure theoretical interest and not practical, it is possible to exploit the huge computing power available on the current multi-core computers to reduce their running time by parallelization and make them feasible to use in practice. We hope that this paper will also increase the interest of the parallel computing community in developing

practical parallel approximation algorithms for other important NP-hard problems. In our future work we plan to design parallel approximation algorithms for other NP-hard problems and eventually develop a set of general techniques that can be used to design such algorithms.

References

[1] D. P. Williamson, D. B. Shmoys, The Design of Approximation Algorithms, Cambridge University Press, 2011.

[2] S. Khuller, U. Vishkin, N. E. Young, A primal-dual parallel approximation technique applied to weighted set and vertex covers, J. Algorithms 17 (2) (1994) 280–289.

[3] S. Rajagopalan, V. V. Vazirani, Primal-dual RNC approximation algorithms for set cover and covering integer programs, SIAM J. Comput. 28 (2) (1998) 525–540.

[4] G. E. Blelloch, R. Peng, K. Tangwongsan, Linear-work greedy parallel approximate set cover and variants, in: Proc. of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2011, pp. 23–32.

[5] G. E. Blelloch, K. Tangwongsan, Parallel approximation algorithms for facility-location problems, in: Proc. of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2010, pp. 315–324.

[6] Q. Wang, K. Cheng, Parallel time complexity of a heuristic algorithm for the k-center problem with usage weights, in: Proc. of the Second IEEE Symposium on Parallel and Distributed Processing, 1990, pp. 254–257.

[7] E. W. Mayr, Parallel approximation algorithms, Tech. Rep. STAN-CS-88-1225, Department of Computer Science, Stanford University, California (September 1988).

[8] D. S. Hochbaum, D. B. Shmoys, Using dual approximation algorithms for scheduling problems: theoretical and practical results, J. ACM 34 (1) (1987) 144–162.

[9] E. Lawler, J. Lenstra, A. Rinnooy Kan, D. Shmoys, Sequencing and scheduling: Algorithms and complexity, in: S. Graves, A. R. Kan, P. Zipkin (Eds.), Handbooks in Operations Research and Management Science: Logistics of Production and Inventory, North-Holland, Amsterdam, 1993.

[10] R. M. Karp, Reducibility among combinatorial problems, in: R. Miller, J. Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, New York, 1972, pp. 85–104.

[11] M. R. Garey, D. S. Johnson, Strong NP-completeness results: Motivation, examples, and implications, J. ACM 25 (3) (1978) 499–508.

[12] R. L. Graham, Bounds for certain multiprocessing anomalies, Bell System Technological Journal 45 (1966) 1563–1581.

[13] R. L. Graham, Bounds on multiprocessing timing anomalies, SIAM J. on Applied Mathematics 17 (2) (1969) 416–429.

[14] E. G. Coffman, M. R. Garey, D. S. Johnson, An application of bin-packing to multiprocessor scheduling, SIAM J. Comput. 7 (1) (1978) 1–17.

[15] D. K. Friesen, Tighter bounds for the multifit processor scheduling algorithm, SIAM J. Comput. 13 (1) (1984) 170–181.

[16] M. Yue, On the exact upper bound for the multifit processor scheduling algorithm, Annals of Operations Research 24 (1) (1990) 233–259.

[17] D. Friesen, M. Langston, Evaluation of a multifit-based scheduling algorithm, J. Algorithms 7 (1) (1986) 35–59.

[18] S. Sahni, Algorithms for scheduling independent tasks, J. ACM 23 (1) (1976) 116–127.

[19] D. Helmbold, E. W. Mayr, Fast scheduling algorithms on parallel computers, Tech. Rep. STAN-CS-84-1025, Department of Computer Science, Stanford University, California (November 1984).

[20] L. Ghalami, D. Grosu, A parallel approximation algorithm for scheduling parallel identical machines, in: Proc. of the 7th IEEE Workshop on Parallel/Distributed Computing and Optimization, 2017, pp. 442–451.

[21] G. Li, B. W. Wah, Systolic processing for dynamic programming problems, in: Proc. of the International Conference on Parallel Processing, 1985, pp. 434–441.

[22] W. Martins, J. D. Cuvillo, F. Useche, K. Theobald, G. Gao, A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison, Pacific Symposium on Biocomputing (6) (2001) 311 – 322.

[23] S. Maleki, M. Musuvathi, T. Mytkowicz, Parallelizing dynamic programming through rank convergence, 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[24] M. Harchol-Balter, A. Downey, Exploiting process lifetime distributions for dynamic load balancing, Proc. SIGMETRICS (1996) 13–24.

[25] C. N. Potts, Analysis of a linear programming heuristic for scheduling unrelated parallel machines., Discrete Applied Mathematics 10 (2) (1985) 155–164.

[26] IBM ILOG CPLEX optimization studio for academics initiative.
URL `http://www-01.ibm.com/software/websphere/products/optimization/academic-initiative/`