A Parallel Approximation Algorithm for Scheduling Parallel Identical Machines

Laleh Ghalami and Daniel Grosu
Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
{laleh.ghalami, dgrosu}@wayne.edu

Abstract—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 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 solve the integer program formulation of the problem.

I. 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 $P || C_{max}$, 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 $P || C_{max}$ problem is defined as follows:

$P || C_{max}$ 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 $P || C_{max}$ 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 $\epsilon$-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 $\epsilon$ from that of the optimal solution. We call $\epsilon$, the approximation ratio of the algorithm. For maximization problems, $\epsilon < 1$, while for minimization problems, $\epsilon > 1$. 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 $\epsilon$ there is an algorithm $A_\epsilon$ 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 $A_\epsilon$ are polynomial on both the problem size and $1/\epsilon$ [1].

Several approximation algorithms, with different approximation guarantees, have been proposed for $P || C_{max}$. 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 $P || C_{\text{max}}$ is strongly NP-hard [11], a PTAS provides the strongest approximation guarantee one could obtain, unless $P=NP$. That is, no FPTAS exists for $P || C_{\text{max}}$, 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.

A. Related Work

Since Karp [10] showed that the $P || C_{\text{max}}$ 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 multijet 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 $1 + 2 + 2^{-k}$. Sahni [15] 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 $P || C_{\text{max}}$, assuring that with a relative error of $\varepsilon$, it executes in time $O((n/\varepsilon)^{1/\varepsilon^2})$. 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 $P || C_{\text{max}}$. Helmbold and Mayr [16] 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 $(1 + \varepsilon)$-approximation algorithm for $P || C_{\text{max}}$ 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.

B. Our Contributions

We address two issues that were not considered in the traditional design of approximation algorithms for the $P || C_{\text{max}}$ 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 $P || C_{\text{max}}$ 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 solve the integer program formulation of the problem.

C. Organization

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

II. A PTAS FOR $P || C_{\text{max}}$

We present the PTAS for $P || C_{\text{max}}$ 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 $t_i$, $i = 1, \ldots, n$; and the relative error $\varepsilon > 0$. We denote by $T$ the multiset of jobs’ processing times, i.e., $T = \{t_1, \ldots, t_n\}$, 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:

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

(1)
Algorithm 1 PTAS for $P||C_{\text{max}}$

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

and

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

(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 $T$ of the processing times of the jobs is partitioned into two multisets: $L$, containing the processing times of the long jobs, where a long job is a job $j$ with $t_j > T/k$; and $S$, 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 $\left[ T/k^2 \right]$ and includes them in the multiset $\tilde{L}$ of rounded processing times 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 $i\left[ T/k^2 \right]$, $i = 1, \ldots, k^2$ (Lines 19-24). After creating 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 $J = \{\tilde{J}_1, \tilde{J}_2, \ldots, \tilde{J}_m\}$ of the long jobs of rounded sizes, where $\tilde{J}_i$ is the set of long jobs assigned to machine $i$.

The $DP$ algorithm generates the set $C$ 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 $i\left[ T/k^2 \right]$ assigned to that machine, and satisfying

\[ \sum_{i=1}^{k^2} i \left[ \frac{T}{k^2} \right] s_i \leq T. \]  

(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:

\[ OPT(n_1, \ldots, n_{k^2}) = 
1 + \min_{(s_1, \ldots, s_{k^2}) \in C} OPT(n_1 - s_1, \ldots, n_{k^2} - s_{k^2}). \]  

(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 $\text{OPT}(n_1, \ldots, n_{k+2})$ are the components of a dynamic programming table. If the $\text{DP}$ 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 $J$. Therefore, we can interpret it as a schedule for the long jobs with original processing times with a makespan at most $(1 + \frac{1}{\epsilon})T$, 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 $\mathbf{J}$ (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 $(1 + \frac{1}{\epsilon})T$, 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 $(1 + \frac{1}{\epsilon})T$ 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

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

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

Since there are $O(nk^2)$ machine configurations that are considered by the algorithm and $k = \lceil 1/\epsilon \rceil$, its running time in the worst case 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 $P \parallel C_{\text{max}}$ is strongly NP-hard [11], and therefore, there does not exist a FPTAS for solving it, unless $P=NP$.

III. PARALLEL APPROXIMATION ALGORITHM

In this section, we introduce our proposed parallel approximation algorithm for $P \parallel C_{\text{max}}$. 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 $\text{DP}$ 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 [17]. 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 $\text{DP}$-table.

The $\text{DP}$ algorithm determines the optimal number of identical parallel machines needed for scheduling $\sum_{i=1}^{k} n_i$ jobs having a makespan within $T$, by computing the value of $\text{OPT}(n_1, \ldots, n_{k+2})$, 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 $i[T/k^2]$, $i = 1, \ldots, k^2$. Therefore, there are $\prod_{i=1}^{k^2} (n_i + 1)$ distinct entries in the $\text{DP}$-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 $O((n/k)^{1/\epsilon}k^2)$. This running time makes $\text{DP}$ the most expensive component of the PTAS and motivates us to consider it for parallelization.

In order to obtain a parallel version of the $\text{DP}$ algorithm we need to investigate the dependencies between the dynamic programming subproblems. Thus, we are interested to know how the $\text{DP}$ algorithm (Algorithm 2) fills out the $\text{DP}$-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:

$$N = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0).$$  

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

$$C = \{ \{0, 0\}, \{0, 1\}, \{0, 2\}, \{1, 0\}, \{1, 1\}, \{1, 2\}, \{2, 0\}, \{2, 1\} \}.$$
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 Equation (4) that we need the following subproblems to calculate $OPT(N)$:

$$\begin{align*}
OPT(2, 3) &= 1 + \min \{OPT(2, 3), OPT(2, 2), \\
&\quad OPT(1, 3), OPT(1, 2), \\
&\quad OPT(1, 1), OPT(0, 3), OPT(0, 2)\}, \\
\end{align*}$$

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

$$
OPT(2, 3) = 1 + \min \{OPT(2, 2), OPT(1, 3), OPT(1, 2), OPT(1, 1), OPT(0, 3), OPT(0, 2)\},
$$

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 I). 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. Figure 1 shows the assignment of the subproblems for the small example presented above to a parallel system composed of four processors. This is the idea we use in the design of our parallel algorithm.

The entries of the $DP$-table (Table I) 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:

$$
V = \{ (0, 0), (0, 1), (0, 2), (0, 3), \\
(1, 0), (1, 1), (1, 2), (1, 3), \\
(2, 0), (2, 1), (2, 2), (2, 3) \}.
$$

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 $\sigma$, 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 I. 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 $OPT(2, 0)$, $OPT(1, 1)$ and $OPT(0, 2)$ (marked with arrows) are as follows:

$$
\begin{align*}
OPT(2, 0) &\leftarrow \{OPT(1, 0), OPT(0, 0)\} \\
OPT(1, 1) &\leftarrow \{OPT(1, 0), OPT(0, 1), OPT(0, 0)\} \\
OPT(0, 2) &\leftarrow \{OPT(0, 1), OPT(0, 0)\},
\end{align*}
$$

(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 =
the sums of the distances of the vectors last and is handled iteratively. Hence, the flow of computation recursively computes the other entries until it ends up at the (Algorithm 2) starts from the last entry of the DP here that the recursive sequential version of the makespan, DP that is, \( \text{OPT}(N) \).

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 \( N = (n_1, \ldots, n_k) \), 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 \( \text{OPT}(N) \). 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 \text{"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 \( \sigma \) 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 \( n_1 + n_2 + \ldots + n_k + 1 = n' + 1 \), 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 \( q_i \) the number of subproblems in each anti-diagonal. Note that the number of the subproblems assigned to each processor depends on \( q_i \). If \( q_i \geq P \) then each of the \( P \) processors compute at most \( \lceil q_i/P \rceil \) subproblems from anti-diagonal \( l \); else \( q_i \) processors out of \( P \), compute the \( q_i \) 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, \( \text{OPT}(0, \ldots, 0) \) 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 \( C_{v^i} \) of all possible machine configurations, \( (s_1, \ldots, s_k) \), for vector \( v^i \) (Line 17). Note that \( C_{v^i} \) does not include the zero vector, since it means no assignment. Next, the algorithm reads the values of all subproblems \( \text{OPT}(v^i - s_1, \ldots, v^i - s_k) \) from the DP-table and places their optimal values into multiset \( O_{v^i} \) (Lines 18-19). Then, it determines the minimum among all values of the subproblems currently in \( O_{v^i} \), adds 1 to the minimum and assigns the value to subproblem \( \text{OPT}(v^i, \ldots, v^i) \) (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 \( \text{OPT}(N) \) is computed and returned (Line 26).

Our proposed parallel approximation algorithm for solving \( P||C_{\text{max}} \) 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

### Algorithm 3 Parallel DP(N,T)

```
1: Input: \( N = (n_1, \ldots, n_k) \), T
2: \( \sigma \leftarrow (n_1 + 1)(n_2 + 1) \ldots (n_k + 1) \)
3: Let \( v^i = (v^i_1, \ldots, v^i_k) \) and \( \text{OPT}(v^i_1, \ldots, v^i_k) \) 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: \( d_i = 0 \)
6: for \( j = 0, \ldots, k^2 - 1 \) do
7: \( d_i = d_i + v^i_j \)
8: end parallel for
9: \( n' = n_1 + \ldots + n_k \)
10: for \( l = 0, \ldots, n' \) do
11: parallel for \( i = 0, \ldots, \sigma - 1 \) do
12: if \( d_i = l \) then
13: if \( i = 0 \) then
14: \( \text{OPT}(0, \ldots, 0) \leftarrow 0 \)
15: break
16: \( O_{v^i} \leftarrow \emptyset \)
17: \( C_{v^i} \leftarrow \text{all machine configurations of vector } v^i \)
18: for all \( (s_1, \ldots, s_k) \in C_{v^i} \) do
19: \( O_{v^i} \leftarrow O_{v^i} \cup \{ \text{OPT}(v^i - s_1, \ldots, v^i - s_k) \} \)
20: \( \text{min} \leftarrow \infty \)
21: for all \( o \in O_{v^i} \) do
22: if \( \text{min} > o \) then
23: \( \text{min} = o \)
24: \( \text{OPT}(v^i_1, \ldots, v^i_k) \leftarrow \text{min} + 1 \)
25: end parallel for
26: return \( \text{OPT}(n_1, \ldots, n_k) \)
```
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.

IV. ANALYSIS OF THE PARALLEL ALGORITHM

We analyze the running time of the parallel approximation algorithm presented in Section III, starting with 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 \( q_l < n^{k^2-1} \) 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 \( O((n^{2})^{(1/2)^{2}} / P) \). This is the dominant component of the running time of the proposed parallel approximation algorithm for \( P \parallel C_{max} \) and it represents its overall runtime complexity.

V. EXPERIMENTAL RESULTS

In this section, we investigate the performance of the proposed parallel approximation algorithm by performing extensive experiments on a multi-core system. 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 integer programing formulation of the problem, and that of two other classical approximation algorithms LPT, and LS.

A. 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 \( m = 10, 20 \) machines, and \( n = 30, 50, 100 \) jobs. We generate the processing times of the jobs from the uniform distribution within different intervals: \( U(1, 2m - 1) U(1, 100) U(1, 10) U(1, 10n) \), where \( U(x, y) \) denotes the uniform distribution within the interval \([x, y] \). Since the processing time of the jobs influence the performance of the proposed parallel approximation algorithm and the PTAS, we selected these intervals to obtain instances with small (e.g., \( U(1, 10) \)) and large (e.g., \( U(1, 10n) \)) processing times for the jobs. We also considered instances in which the processing times of the jobs depend on the number of machines and the number of jobs (e.g., \( U(1, 2m - 1) \) and \( U(1, 10n) \)). Therefore, we consider 24 types of instances and for each type we generate 20 different instances, for a total of 480 instances.

We implemented the parallel approximation algorithm in C++ and OpenMP. The parallel approximation algorithm and the sequential PTAS are executed with \( \epsilon = 0.3 \). We choose this value of \( \epsilon \) to obtain an approximation ratio for our algorithm that is below the approximation ratio of LPT. We also implemented three other sequential approximation algorithms, PTAS, LPT and LS in C++. We solve the integer program formulation of the problem by using the CPLEX solver provided by IBM ILOG CPLEX Optimization Studio for Academics Initiative [18] and obtain the optimal solution. We denote by IP (Integer Program) the CPLEX-based implementation that obtains the optimal solution. We run all the algorithms on a 16-core 2.4 GHz dual-processor Intel system with 96 GB of RAM.

B. 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 IP is defined as the ratio of the running time of the IP (i.e., the CPLEX-based) algorithm 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, in our case, the actual approximation ratio of an approximation algorithm for \( P \parallel C_{max} \) is the ratio of the makespan obtained by the algorithm and the optimal makespan obtained by IP.
Figure 2(a) 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 16 for all four types of instances with 20 machines and 100 jobs. The values of the speedup for each type of instance are the averages over the speedup for 20 instances of that type. The proposed parallel algorithm achieves significant average speedup of up to 6.5 using 8 cores and up to 11.7 using 16 cores for \( U(1, 10) \) type instances. This shows that the parallel algorithm is scalable for the sizes of the multicore system used in the experiments. For the problem size considered in this experiment, we expect the speedup to exhibit smaller increases as the number of cores increases past 16. Figure 2(b) shows the average speedup with respect to IP. The proposed algorithm achieves significant speedup of up to 15, using 16 cores with respect to the IP, the CPLEX-based solution. Relative small speedup is obtained by the parallel algorithm for instances of type \( U(1, 10) \) and \( U(1, 10n) \). CPLEX was able to find a solution in a smaller amount of time for these instances compared to the the other
two types of instances. We do not have enough details on the internal CPLEX implementation to be able to explain why this happened for those type of instances. Figure 2(c) shows the average running times for all four types of instances considered in the experiment. For the $U(1, 100)$ type instances IP required about 65 seconds to produce the solution, while our parallel algorithm required only 2.9 seconds on 16 cores.

We now analyze the performance of our algorithms for the same type of instances but for smaller number of machines and fewer jobs. One of these instances ($m = 10$ and $n = 50$) represent the best case for our parallel algorithm when we consider the speedup with respect to IP (i.e., CPLEX based solution). Figure 3(a) and Figure 3(b) show the average speedup obtained by the proposed parallel approximation algorithm with respect to the PTAS and the IP, respectively. The proposed parallel algorithm achieves good average speedup with respect to the PTAS, of up to 5.8 using 16 cores for instances $U(1, 10)$ and $U(1, 2m − 1)$. The proposed algorithm achieves significant speedup of up to 800, using 16 cores with respect to IP for instances $U(1, 10n)$. This large speedup is obtained because CPLEX required a large amount of time to find the optimal solution. Figure 3(c) shows the running times for all four types of instances. For the $U(1, 10n)$ instances IP requires about 105 seconds to produce the solution while our algorithm on 16 cores requires only 0.1 seconds.

We now present the results for a set of instances that in a sense represent the worst cases for our parallel algorithm when we consider the speedup with respect to IP (i.e., CPLEX based solution). Figure 4(a) shows the average speedup obtained by the proposed parallel approximation algorithm with respect to the sequential PTAS for all four types of instances with 10 machines and 30 jobs. For the instance with the large range for the uniform distribution of job processing times, $U(1, 10n)$, our algorithm achieves significant speedup with respect to IP as shown in Figure 4(b). The speedup achieved for the other instances is not as good ranging from 2.0 to 3.7 on 16 cores, while no speedup is achieved for 2 cores. This behaviour is not due to our algorithm but is mainly due to the CPLEX solver being able to solve much more efficiently the integer programs associated with those instances.

We now analyze the actual approximation ratios obtained by the algorithms considered in our experiments, where the actual approximation ratio is defined as the ratio of the makespan obtained by the corresponding algorithm (parallel approximation algorithm, LPT, LS) and the makespan obtained by IP, the CPLEX-based solution which produces the optimal schedule and makespan. Out of all the instances in our experiments we select the instances that represent the best and worst cases for our parallel approximation algorithm with respect to the actual approximation ratios. For all other instances in our experiments our parallel approximation algorithm obtains actual approximation ratios at least as good as those of LPT. Table II and Table III show the best and worst case instances. To analyze the best and worst cases we also considered a special type of instance that is very close to the worst case for the LPT algorithm [13], that is, the number of jobs is $n = 2m + 1$ and the jobs sizes are drawn from $U(m, 2m − 1)$. We also consider a type of instance that has a narrow range
for the processing time of the jobs, that is, \( U(95, 105) \). Figures 5 (a) and (b) show the actual approximation ratios obtained by all four algorithms. Since the actual approximation ratios obtained by the sequential PTAS are the same as those obtained by our parallel approximation algorithm, in Figures 5 (a) and (b) we show only the ratio for the parallel approximation algorithm. In the best case (instance I6) the difference between the approximation ratio of LPT and that of the parallel approximation algorithm is 0.28 (the ratio obtained by LPT is 1.28 while the ratio for our algorithm is 1). In all the best cases our algorithm obtains a ratio under 1.1, much lower than the error \( \epsilon + 1 = 1.3 \) used in the experiments with our algorithm. In the worst cases, the difference between the approximation ratios of our parallel approximation algorithm and that of LPT is at most 0.13, much lower than the difference obtained in the best cases. As expected in all these cases LS obtains the worst approximation ratios among all algorithms. Our parallel approximation algorithm obtains much lower actual approximation ratios than the chosen \( \epsilon \) and for almost all instances obtains actual approximation ratios at least as good as LPT.

VI. 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 multi-core system, and analyzed its performance. 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 program formulation of the problem, and that of two other classical approximation algorithms LPT, and LS. 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. 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