Accepted Manuscript

Scheduling for energy minimization on restricted parallel processors

Xibo Jin, Fa Zhang, Liya Fan, Ying Song, Zhiyong Liu

PII: S0743-7315(15)00064-7
DOI: http://dx.doi.org/10.1016/j.jpdc.2015.04.001
Reference: YJPDC 3393

To appear in: *J. Parallel Distrib. Comput.*

Received date: 29 April 2014
Revised date: 4 March 2015
Accepted date: 1 April 2015

Please cite this article as: X. Jin, F. Zhang, L. Fan, Y. Song, Z. Liu, Scheduling for energy minimization on restricted parallel processors, *J. Parallel Distrib. Comput.* (2015), http://dx.doi.org/10.1016/j.jpdc.2015.04.001

Scheduling for Energy Minimization on Restricted Parallel Processors

Xibo Jin[a,d,*], Fa Zhang[b], Liya Fan[e], Ying Song[c], Zhiyong Liu[a,c]

[a] *Beijing Key Laboratory of Mobile Computing and Pervasive Device, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*
[b] *Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*
[c] *State Key Laboratory for Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*
[d] *University of Chinese Academy of Sciences, Beijing, China*
[e] *IBM China Research Laboratory, Beijing, China*

Abstract

Scheduling for energy conservation has become a major concern in the field of information technology because of the need to reduce energy use and carbon dioxide emissions. Previous work has focused on the assumption that a task can be assigned to any processor. In contrast, we initially study the problem of task scheduling on restricted parallel processors. The restriction takes account of affinities between tasks and processors; that is, a task has its own eligible set of processors. We adopt the Speed Scaling (SS) method to save energy under an execution time constraint (on the makespan $C_{\max}$), and the processors can run at arbitrary speeds in $[s_{\min}, s_{\max}]$. Our objective is to minimize the overall energy consumption. The energy-efficient scheduling problem, involving task assignment and speed scaling, is inherently complex as it is proved to be NP-complete for general tasks. We formulate the problem as an Integer Programming (IP) problem. Specifically, we devise a polynomial-time optimal scheduling algorithm for the case in which tasks have a uniform size. Our algorithm runs in $O(mn^3 \log n)$ time, where m is the number of processors and n is

*Corresponding author
Email addresses: jinxibo@ict.ac.cn (Xibo Jin), zhangfa@ict.ac.cn (Fa Zhang), fanliya@cn.ibm.com (Liya Fan), songying@ict.ac.cn (Ying Song), zyliu@ict.ac.cn (Zhiyong Liu)

the number of tasks. We then present a polynomial-time algorithm that achieves a bounded approximation factor when the tasks have arbitrary-size work. Numerical results demonstrate that our algorithm could provide an energy-efficient solution to the problem of task scheduling on restricted parallel processors.

Keywords: Energy-efficient scheduling; restricted parallel processors; speed scaling; continuous speed model; approximation algorithm.

1. Introduction

Energy consumption has become an important issue for today's computational systems. Dynamic speed scaling is a popular approach to energy-efficient scheduling. It significantly reduces energy dissipation by dynamically changing the speeds of the processors. It is well known that speed and power are related by a cube-root rule. More precisely, a processor consumes power at a rate proportional to s^3 when it runs at a speed s [1, 2]. Most research publications [3, 4, 5, 6, 7, 8, 9, 10] have assumed a more general power function s^α, where $\alpha > 1$ is a constant power parameter. Note that the power is a convex function of the processor speed. Obviously, the energy consumption is the power integrated over time. Higher speeds allow faster execution, but at the same time result in higher energy consumption.

In the past few years, energy-efficient scheduling has received much attention for both single-processor and parallel-processor environments. In the algorithm community, the approaches used can generally be categorized into the following two classes with respect to reducing energy usage [5, 7]:

1. *Dynamic speed scaling.* The processors lower their speeds as much as possible in such a way that they can still execute tasks while fulfilling the time constraints on those tasks. The reason why energy is saved via this strategy is the convexity of the power function. The goal is to decide the processing speeds in a way that minimizes the total energy consumption and guarantees the prescribed deadline.

2. *Power-down management.* The processors are put into a power-saving state when they are idle. However, there is an energy cost of the transition back to the active state. In this strategy, one determines whether there exist idle periods that can outweigh the transition cost and decides when to wake processors from the power-saving mode in order to complete all tasks in time.

Our paper focuses on energy-efficient scheduling via the dynamic speed scaling strategy. In this policy, the goals of scheduling are either to minimize the total energy consumption or to trade off the conflicting objectives of energy and performance. The main difference is that the former goal reduces the total energy consumption as long as the time constraint is not violated, whereas the latter seeks the best point between the energy cost and some performance metrics (such as the makespan and flow time).

Intensive research, initiated by Yao *et al.* [3], has been done on saving energy by speed scaling. In previous work, it was assumed that a task can be assigned to any processor. But it is natural to consider restricted scheduling in modern computational systems. The reason is that systems have evolved over time, for example by the use of clusters of processors, so that the various processors in a system may differ from each other in their abilities. (For instance, processors may have different additional components or different memory capacities [11].) This means that a task can only be assigned to a processor that has the components required for that task. That is, there are different affinities between tasks and processors. In practice, certain tasks may have to be allocated to certain physical resources (such as graphics processing units) [12]. It has also been pointed out that the design of some processors is specialized for particular types of tasks, and therefore tasks should be assigned to the processor best suited for them [13]. Furthermore, when tasks and input data are considered, tasks need to be assigned to the processors that contain their input data (by means of Hadoop Data Locality-Aware, for instance [14]). In other words, some of the tasks can be assigned to a processor set A_i, and some of the tasks to a processor set A_j, but $A_i \neq A_j$, $A_i \cap A_j \neq \emptyset$. Another case in point is scheduling with processor restrictions aimed at minimizing the makespan. This case has been studied extensively; see [11] for an excellent survey. Therefore, it is important to study scheduling with processor restrictions for reasons of both practical and algorithmic requirements.

Our contribution: In this paper, we address the problem of task **S**cheduling with the objective of **E**nergy **M**inimization on **R**estricted **P**arallel **P**rocessors (SEMRPP). We assume that all tasks are ready at the beginning of the process and share a common deadline (a real-time constraint) [2, 4, 6, 7]. We discuss a continuous speed setting where the processors can run at arbitrary speeds in $[s_{\min}, s_{\max}]$. Our main contributions can be summarized in the following three groups:

1. We propose an optimal scheduling algorithm for the case when all of the tasks have uniform computational work.
2. For the general case in which the tasks have nonuniform computational work, we prove that the minimization of energy is NP-complete in the strong sense. We give a $2^{\alpha-1}(2 - 1/p^{\alpha})$-approximation algorithm, where α is the power parameter and $p = \max_{\mathcal{M}_j} |\mathcal{M}_j|$, and where $\mathcal{M}_j$ is the eligible processing set for the task J_j.
3. The performance of the approximation algorithm is evaluated by a set of simulations after an analysis of the algorithm, and it is found that the simulation results are consistent with the proposed scheduling algorithm.

To the best of our knowledge, our work may be the first attempt to study energy consumption optimization with restricted parallel processors.

The remainder of this paper is organized as follows. Section II describes previous work on speed scaling. Section III provides a formal description of the model. Section IV first discusses some preliminary results and formulates the problem as an integer programming problem. Then we devise a polynomial-time optimal scheduling algorithm in the case where the tasks have uniform size, and present a bounded-factor approximation algorithm for the general case in which the tasks have arbitrary-size work. Section V presents numerical results. Finally, we conclude the paper in Section VI.

2. Related Work

Yao *et al.* [3] were the first to explore the problem of scheduling a set of tasks with the least amount of energy in a single-processor environment via speed scaling. They proposed an optimal offline greedy algorithm and two bounded online algorithms, named *Optimal Available* and *Average Rate*. Ishihara *et al.* [4] formulated the problem of energy minimization in dynamical voltage scheduling as an integer linear programming problem where all tasks were ready at the beginning and shared a common finishing time. They showed that in the optimal solution a processor runs at only two adjacent discrete speeds when it can use only a small number of discrete processor speeds.

Besides studying variants of the speed scaling problem on a single processor, researchers have also carried out studies on parallel-processor environments. Chen *et al.* [6] considered energy-efficient scheduling with and without task migration in a multiprocessor system. They proposed an approximation algorithm for different settings of the power characteristics where no task was allowed to migrate. When task migration was allowed and the migration cost was assumed to be negligible, they showed that there was an optimal real-time task-scheduling algorithm. Albers *et al.* [7] investigated the basic problem of scheduling a set of tasks in a multiprocessor setting with the objective of minimizing the total energy consumption. First, they studied the case in which all tasks have unit size, and proposed a polynomial-time algorithm for agreeable deadlines. They proved that this case is NP-hard for arbitrary release times and deadlines and gave an $\alpha^{\alpha}2^{4\alpha}$-approximation algorithm. For scheduling tasks with arbitrary processing size, they developed constant-factor approximation algorithms. Aupy *et al.* [2] studied the minimization of energy for a set of processors for which a task assignment had been given, and investigated different speed scaling models. Angel *et al.* [10] considered a multiprocessor migratory and preemptive scheduling problem with the objective of minimizing the energy consumption. They proposed an optimal algorithm in the case where the jobs have release dates, deadlines, and a power parameter $\alpha > 2$.

There are also some publications that describe research on performance with an energy bound. Pruhs *et al.* [8] discussed the problem of speed scaling to optimize the makespan under the constraint of an energy budget in a multiprocessor environment where the tasks had precedence constraints ($Pm|\text{prec}, \text{energy}|C_{\max}$, where m is the number of processors). They reduced the problem to $Qm|\text{prec}|C_{\max}$ and obtained a poly-log(m)-approximation algorithm assuming that the processors can change speed continuously over time. Greiner *et al.* [9] presented research on the trade-off between energy and delay; i.e., their objective was to minimize the sum of the energy cost and delay cost. They suggested a randomized algorithm $\mathcal{RA}$ for multiple processors: each task was assigned uniformly at random to a processor, and then the single-processor algorithm $\mathcal{A}$ was applied separately to each processor. They proved that the approximation factor for $\mathcal{RA}$ was βB_α without task migration when $\mathcal{A}$ was a β-approximation algorithm (here, B_α is the α-th Bell number). They also showed that any β-competitive online algorithm for a single processor yields a randomized βB_α-competitive online algorithm for multiple processors without migration. Using the method of conditional expectations, the results could be transformed to a derandomized version with additional running time. Angel *et al.* [10] also extended their algorithm by considering minimizing the energy consumption, so as to obtain an optimal algorithm for the problem of maximum-lateness minimization under the constraint of an energy budget.

However, all of these results were established without taking restricted parallel processors into account. More formally, let the set of tasks $\mathcal{J}$ and the set of processors $\mathcal{P}$ construct a bipartite graph $G = (\mathcal{J} + \mathcal{P}, E)$, where an edge of E denotes that a task can be assigned to a processor. In previous work, G was a *complete* bipartite graph, i.e., for any two vertices $v_1 \in \mathcal{J}$ and $v_2 \in \mathcal{P}$, the edge $v_1 v_2$ is in G. We study the problem of energy-efficient scheduling in which G is a *general* bipartite graph, i.e., $v_1 v_2$ need not be an edge of G.

We emphasize that, as stated in recent reports [15, 16], every year the energy costs of computer systems are on the order of billions of dollars. Given this, a reduction in the energy costs by a small percentage could result in savings of billions of dollars.

3. Problem and Model

We model the SEMRPP problem of scheduling a set $\mathcal{J} = \{J_1, J_2, \dots, J_n\}$ of n independent tasks on a set $\mathcal{P} = \{P_1, P_2, \dots, P_m\}$ of m processors. Each task J_j has an amount of computational work w_j, which is defined as the number of CPU cycles required for the execution of J_j [3]. We refer to the set $\mathcal{M}_j \subseteq \mathcal{P}$ as the eligible processing set for the task J_j; that is, J_j needs to be scheduled on one of its eligible processors $\mathcal{M}_j$ ($\mathcal{M}_j \neq \emptyset$). We also say that J_j is allowable on processor $P_i \in \mathcal{M}_j$, and is not allowed to migrate after it has been assigned to a processor (it is nonmigratory). A processor can process at most one task at a time, and all processors are available at the beginning of the operation.

At any time t, the speed of J_j is denoted as s_{jt}, and the corresponding processing power is $(s_{jt})^\alpha$. The number of CPU cycles w_j executed in a time interval is the speed integrated over time, and the energy consumption E_j is the power integrated over time; that is, $w_j = \int s_{jt}\, dt$ and $E_j = \int (s_{jt})^\alpha\, dt$, following the classical models in the literature [2, 3, 4, 5, 6, 7, 8, 9, 10]. Note that in this work we focus on speed scaling and all processors are alive during the whole execution, and so we do not take static energy into account [2, 7, 8, 9]. Let c_j be the time when the task J_j finishes its execution. Let x_{ij} be a 0–1 variable which is equal to one if task J_j is processed on processor P_i and zero otherwise. Note that $x_{ij} = 0$ if $P_i \notin \mathcal{M}_j$. Our goal is to schedule the tasks on the processors to minimize the overall energy consumption, where each task must finish before the given common deadline C and be processed on its eligible processors. The SEMRPP problem is then formulated as follows:

$$(\mathbf{P}_0) \qquad \min \sum_{j=1}^{n} \int (s_{jt})^\alpha\, dt$$

$$
\begin{aligned}
\text{s.t.} \qquad c_j \le C \quad \forall J_j,\\
\sum_{i=1}^{m} x_{ij} = 1 \quad \forall J_j,\\
x_{ij} \in \{0,1\} \quad \forall J_j, P_i \in \mathcal{M}_j,\\
x_{ij} = 0 \quad \forall J_j, P_i \notin \mathcal{M}_j.
\end{aligned}
$$

4. Algorithms and Analysis

In this section, we start by giving preliminary lemmas so that we can reformulate the SEMRPP problem. After that, we present an exact algorithm using the maximum flow to deal with the situation where the tasks have a uniform size, and give a proof of correctness. Finally, we seek a polynomial-time approximation algorithm with a constant bounded factor for the general case in which tasks have different numbers of execution cycles.

4.1. Preliminary Lemma

Lemma 1. *If S is an optimal schedule for the SEMRPP problem in the continuous model, it is optimal to execute each task at a unique speed throughout its execution.*

Proof. Suppose S is an optimal schedule in which some task J_j does not run at a unique speed during its execution. We denote J_j's speeds by $s_{j1}, s_{j2}, \ldots, s_{jk}$, the power for each speed i is $(s_{ji})^\alpha, i = (1, 2, \ldots, k)$, and the execution times for these speeds are $t_{j1}, t_{j2}, \ldots, t_{jk}$, respectively. So, the energy consumption is $\sum_{i=1}^{k} t_{ji}(s_{ji})^\alpha$. We average the k speeds and keep the total execution time unchanged, i.e., $\bar{s}_j = (\sum_{i=1}^{k} s_{ji}t_{ji})/(\sum_{i=1}^{k} t_{ji})$. Because the power function is a convex function of the speed, we have the following result because of Jensen's inequality [17] and convexity [18]:

$$\begin{aligned}\sum_{i=1}^{k} t_{ji}(s_{ji})^{\alpha} &= \left(\sum_{i=1}^{k} t_{ji}\right)\left(\sum_{i=1}^{k} \frac{t_{ji}}{\sum_{i=1}^{k} t_{ji}}(s_{ji})^{\alpha}\right) \\ &\geq \left(\sum_{i=1}^{k} t_{ji}\right)\left(\sum_{i=1}^{k} \frac{t_{ji}s_{ji}}{\sum_{i=1}^{k} t_{ji}}\right)^{\alpha} = \left(\sum_{i=1}^{k} t_{ji}\right)(\bar{s}_j)^{\alpha} \\ &= \sum_{i=1}^{k} t_{ji}(\bar{s}_j)^{\alpha}.\end{aligned}$$

(In the rest of the paper, we shall use convexity in many places but will not repeatedly cite reference [18].) So, the energy consumption for a unique speed is less than that for a task run at different speeds. That is, if we do not change J_j's execution time and its assigned processor (satisfying the restriction), we can obtain a schedule with less energy consumption, which contradicts the assumption that S is an optimal schedule. Note that this perspective has also been mentioned in [2]. $\square$

Corollary 1. *There exists an optimal solution to SEMRPP in the continuous model in which each processor executes all tasks at a uniform speed and finishes its tasks at time C.*

The case where all tasks on a processor run at a unique speed can be proved like Lemma 1. If some processor finishes its tasks earlier than C, it can lower its speed to consume less energy without breaking the time constraint and the restriction. Furthermore, there will be no gaps in the schedule [8].

The above discussion leads to a reformulation of the SEMRPP problem in the continuous model as follows:

$$(\mathbf{P}_1) \qquad \min \sum_{i=1}^{m} \frac{\left(\sum\limits_{j=1}^{n} x_{ij}w_j\right)^{\alpha}}{C^{\alpha-1}}$$

$$
\begin{aligned}
\text{s.t.} \qquad \sum_{j=1}^{n} x_{ij} w_j \le s_{\max} C \quad \forall P_i, && (1)\\
\sum_{i=1}^{m} x_{ij} = 1 \quad \forall J_j, && (2)\\
x_{ij} \in \{0,1\} \quad \forall J_j, P_i \in \mathcal{M}_j, && (3)\\
x_{ij} = 0 \quad \forall J_j, P_i \notin \mathcal{M}_j. && (4)
\end{aligned}
$$

The objective function is obtained from assuming that processor P_i runs at a speed $(\Sigma_{J_j \text{ on } P_i} w_j)/C = (\Sigma_{j=1}^{n} x_{ij} w_j)/C$; that is, each task on P_i will run at this speed, and P_i will complete all the tasks on it at time C. (This assumes that, in each problem instance, the number of computational cycles for the tasks on one processor is enough to ensure that the processor will not run at a speed $s_i < s_{\min}$. Otherwise, we are likely to turn off some processors.) Constraint (1) follows, since a processor cannot run at a speed higher than $s_{\max}$. Constraint (2) relates to the fact that if a task has been assigned to a processor it will not be assigned to other processors, i.e., it is nonmigratory. Constraints (3) and (4) are the restrictions of the tasks to particular processors.

Lemma 2. *Finding an optimal schedule for the SEMRPP problem in the continuous model is NP-complete in the strong sense.*

Proof. First, we transform the optimization problem to an associated decision problem: given restrictions on the time and the eligible processors, and a bound on the energy consumption, is there a schedule such that the restrictions and the bound on energy consumption are satisfied? Clearly, this problem is in NP, since we can verify in polynomial time that a proposed schedule satisfies the given restrictions and the bound on energy consumption. We will prove that finding an optimal schedule for the SEMRPP problem is NP-complete in the strong sense via reduction from the 3-partition problem.

Consider an instance $\mathcal{I}_1$ of the 3-partition problem: Given a list $A = (a_1, a_2, \ldots, a_{3m})$ of $3m$ positive integers such that $\sum a_j = mB$, is there a partition of A into m subsets $A_1, A_2, \ldots, A_m$ such that $\sum_{a_j \in A_i} a_j = B$ for each

$1 \leq i \leq m$ [19, 20]? We construct an instance $\mathcal{I}_2$ of the SEMRPP problem as follows: (1) there are m processors $\mathcal{P} = \{P_i\}$, and $s_{\max}$ is fast enough to ensure a feasible schedule for the given tasks; (2) there are $3m$ tasks $\mathcal{J} = \{J_j\}$, for which the numbers of execution cycles w_j are equal to a_j and $\mathcal{M}_j = \mathcal{P}$ for all tasks J_j; and (3) the deadline is $C = 1$ and the energy consumption is mB^α.

Suppose $\mathcal{I}_1$ has a solution; then the tasks $\{J_J : w_j \in A_i\}$ are assigned to processor P_i. So, the energy consumption is $\sum_{i=1}^{m}(\sum_{J_j : w_j \in A_i} a_j)^\alpha / 1^{\alpha-1} = mB^\alpha$. Thus $\mathcal{I}_2$ has a solution.

Suppose $\mathcal{I}_2$ has a solution, and we denote the numbers of execution cycles of the processors by $\{h_1, h_2, \ldots, h_m\}$. According to ($\mathbf{P}_1$), the energy consumption is $\sum_{i=1}^{m}(h_i)^\alpha / 1^{\alpha-1}$. By convexity, we have $\sum_{i=1}^{m}(h_i)^\alpha = m\sum_{i=1}^{m}(1/m)(h_i)^\alpha \geq m((1/m)\sum_{i=1}^{m} h_i)^\alpha = mB^\alpha$. (Note that $\sum_{i=1}^{m} h_i = mB$.) The energy consumption is equal to mB^α if and only if $h_1 = h_2 = \ldots = h_m = B$. Thus $\mathcal{I}_1$ has a solution.

So, we can conclude that SEMRPP in the continuous model is strongly NP-complete by this polynomial-time reduction from the 3-partition problem, which has been proved NP-complete in the strong sense. □

Lemma 3. *There exists a polynomial-time approximation scheme (PTAS) for the SEMRPP problem in the continuous model when $\mathcal{M}_j = \mathcal{P}$ and $s_{\max}$ is fast enough.*

Proof. The proof is somewhat similar to that in [8], whose aim was to give a PTAS for the problem of measuring the makespan under the condition of an energy bound ($Sm|\text{energy}|C_{\max}$). Considering that $\mathcal{M}_j = \mathcal{P}$ and the load of each processor consists of a vector, it turns out that the SEMRPP problem is equivalent to minimizing the l_α norm[1] of the loads [21]. This is concluded from the proof of Lemma 2; that is, if we denote the numbers of execution cycles of the processors by $\{h_1, h_2, \ldots, h_m\}$, the energy consumption is $\sum_{i=1}^{m}(h_i)^\alpha / C^{\alpha-1}$.

[1] For a positive number $\alpha \geq 1$, the l_α norm of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is defined by $\|\mathbf{x}\| = (|x_1|^\alpha + |x_2|^\alpha + \ldots + |x_n|^\alpha)^{1/\alpha}$.

See the part referring to $\sum_{i=1}^{m}(h_i)^\alpha$ and note that α is a constant power parameter. We then use the PTAS given in [21]; that is, for any $\epsilon > 0$, we can find the sum of the numbers of execution cycles of the tasks on the processors P_i (referred to below as the load) $\{L_1, L_2, \ldots, L_m\}$ in polynomial time such that $\Sigma_{i=1}^{m}(L_i)^\alpha \leq (1+\epsilon)\Sigma_{i=1}^{m}(OPT_i)^\alpha$, where L_i is the load of scheduling and OPT_i is the optimal load for processor P_i. $\square$

Note that we have given detailed proofs of Lemmas 2 and 3 similar to those stated in [7], but we have stated mainly the conditions that apply in a restricted environment (such as in the case of the set of restricted processors and the upper speed $s_{\max}$ that we discuss below in the paper).

4.2. Uniform Tasks

We now propose an optimal algorithm for a special case of the SEMRPP problem in which all tasks have equal numbers of execution cycles (uniform) in the continuous model; we denote this algorithm as ECSEMRPP_Algo. Note that in Lemma 2, when we prove the complexity of the SEMRPP problem, the tasks have arbitrary-size work. So it is not contradictory that we give the polynomial-time algorithm for uniform tasks. Without loss of generality, we can set $w_j = 1, \forall J_j$ and set $C = C/w_j$. Given a set of tasks $\mathcal{J}$, a set of processors $\mathcal{P}$, and sets of eligible processors for tasks $\{\mathcal{M}_j\}$, we construct a network $G = (V, E)$ as follows: the vertex set of G is $V = \mathcal{J} \cup \mathcal{P} \cup \{s, t\}$ (s and t correspond to a virtual source and a virtual sink, respectively), and the edge set E of G consists of three subsets: (1) (s, P_i) for all $P_i \in \mathcal{P}$, (2) (P_i, J_j) for $P_i \in \mathcal{M}_j$, and (3) (J_j, t) for all $J_j \in \mathcal{J}$. We set the capacity of the edges (P_i, J_j) and (J_j, t) to unity, and set (s, P_i) to have a capacity c (initially, we can set $c = n$). We define $L^* = \min\{\max\{L_i\}\}$ $(i = 1, 2, \ldots, m)$ as the minimum–maximum load, where L_i is the load of processor P_i; this can be obtained by means of Algorithm 1.

We constructed our algorithm ECSEMRPP_Algo (see Algorithm 5) to find the optimal schedule for the SEMRPP problem where the tasks have uniform size. There are four subprocedures in this algorithm, the main functions of which are as follows:

Algorithm 1: Min_Max_Assign(G, n)

input : (G, n) /* n is the number of tasks */

output: The minimum–maximum load L^* of P_i for all $P_i \in \mathcal{P}$, and the resulting configuration G_H

begin

  Let variable $l = 1$ and variable $u = n$;

  while $l \neq u$ **do**

    Let capacity $c = \lfloor \frac{1}{2}(l + u) \rfloor$. Find the maximum flow in the network G;

    if *The value of the maximum flow is exactly n, namely $L^* \leq c$,* **then**

      set $u = c$ and keep the configuration of maximum flow G^H;

    else

      The value of the maximum flow is less than n, namely $L^* > c$, set $l = c + 1$;

    end

  end

  The optimal value is $L^* = l$, return L^* and G^H;

end

- Min_Max_Assign (see Algorithm 1). Find a "minimum–maximum load" assignment and obtain the minimum–maximum load c^*.

- Pre_Assign (see Algorithm 2). Find an assignment where the capacity of the edges (s, P_i) is a fixed integer. We set this to the minimum–maximum load minus one, i.e., $c^* - 1$.

- Find_Candidate_Critical (see Algorithm 3). Find the processors that have the potential to be assigned one more task if the capacities of the edges (s, P_i) are increased by one, i.e., from $c^* - 1$ to c^*. We define these processors as candidate "actual maximum-load processors" in Definition 1. (We shall also refer to them as "candidate critical processors.")

- Match (see Algorithm 4). Find the actual maximum-load processors from the candidate critical processors, and find a matching between the actual maximum-load processors and their tasks.

Definition 1. *Given that the capacity of the edges (s, P_i) is $c^* - 1$ and given an assignment resulting from this, we say that a processor is a candidate actual maximum-load processor if its load is equal to $c^* - 1$ and it has the potential ability to be given unassigned tasks if the capacity of the edges (s, P_i) is c^*.*

Lemma 4. *The algorithm Min_Max_Assign solves the problem of minimizing the maximum load of the processors for restricted parallel processors in $O(n^3 \log n)$ time if all tasks have equal numbers of execution cycles.*

The proof follows mainly from consideration of the maximum flow as described in [22]. In the Algorithm 1, we use a binary search to decide the minimum–maximum load L^* of P_i for all $P_i \in \mathcal{P}$ when the maximum flow is no less than n. As the range is $[1, n]$, so there is $\log n$ steps. The computational complexity is then equal to the time $O(n^3)$ required to find the maximum flow multiplied by $\log n$ steps, i.e., $O(n^3 \log n)$.

Next, we show some properties of the result of the algorithm Find_Candidate_Critical as follows.

Algorithm 2: Pre_Assign(G, c_{fix})

input : (G, c_{fix})

output: An assignment G^A of the fixed capacity c_{fix}

begin

  Set the capacity of the edges (s, P_i) to c_{fix}, and run the maximum-flow algorithm in the network G. Denote the configuration of the result of the algorithm as G^A;

  Return G^A.

end

Algorithm 3: Find_Candidate_Critical(G, G^A, c_{fix})

input : (G, G^A, c_{fix})

output: The set of candidate actual maximum-load processors $\mathcal{P}^C$, and the set of unassigned tasks $\mathcal{J}^C$

begin

  Compare the network G and the preliminary assignment G^A, and denote the rest of the unassigned tasks as $\mathcal{J}^C$. Find the set of processors $\mathcal{P}^1$ to which the tasks in $\mathcal{J}^C$ can be assigned;

  Set $\mathcal{P}^2 = \emptyset$;

  Set $\mathcal{P}^{'} = \mathcal{P}^1$;

  Find the set of processors $\mathcal{P}^{''}$ to which the tasks currently assigned to $\mathcal{P}^{'}$ can be assigned, except for $(\mathcal{P}^1 \cup \mathcal{P}^2)$;

  while $\mathcal{P}^{''} \neq \emptyset$ **do**

    $\mathcal{P}^2 = \mathcal{P}^2 \cup \mathcal{P}^{''}$;

    Set $\mathcal{P}^{'} = \mathcal{P}^{''}$;

    Find the set of processors $\mathcal{P}^{''}$ to which the tasks currently assigned to $\mathcal{P}^{'}$ can be assigned, except for $(\mathcal{P}^1 \cup \mathcal{P}^2)$;

  end

  Return $\mathcal{P}^C = \mathcal{P}^1 \cup \mathcal{P}^2$ and $\mathcal{J}^C$.

end

Lemma 5. *The unassigned tasks $\mathcal{J}^C$ ($\mathcal{J}^C \neq \emptyset$) can only be assigned to the set of candidate actual maximum-load processors $\mathcal{P}^C$, which is defined in Definition 1. The load of each processor in $\mathcal{P}^C$ is $c^* - 1$.*

Proof. According to the algorithm, $\mathcal{P}^1$ consists of all the processors that the unassigned tasks $\mathcal{J}^C$ can be assigned to. As $\mathcal{P}^1 \subseteq \mathcal{P}^C$, the unassigned tasks $\mathcal{J}^C$ can only be assigned to the processors $\mathcal{P}^C$.

The load of each processor in $\mathcal{P}^1$ is $c^* - 1$ because there are some unassigned tasks that can be allocated to them. Suppose there is a processor P_e in $\mathcal{P}^2$ whose load is less than $c^* - 1$. Then some tasks assigned to the processors in $\mathcal{P}^1$ can be reassigned to P_e, so some of the unassigned tasks can be assigned to the processors in $\mathcal{P}^1$. This contradicts the fact that Algorithm 2 finds a stable maximum-flow assignment. The assumption is wrong; we have the result that the load of each processor in $\mathcal{P}^2$ is $c^* - 1$. As $\mathcal{P}^C = \mathcal{P}^1 \cup \mathcal{P}^2$, we conclude that the load of each processor in $\mathcal{P}^C$ is $c^* - 1$. □

Lemma 6. *The algorithm Find_Candidate_Critical finds the set of candidate actual maximum-load processors $\mathcal{P}^C$, and its size is no less than the size of the unassigned tasks $\mathcal{J}^C$, i.e., $|\mathcal{P}^C| \geq |\mathcal{J}^C|$.*

Proof. The correctness of finding $\mathcal{P}^C$ follows from Definition 1 and the algorithm Find_Candidate_Critical. Now we prove that $|\mathcal{P}^C| \geq |\mathcal{J}^C|$. Suppose we have $|\mathcal{P}^C| < |\mathcal{J}^C|$. According to Lemma 5, the number of tasks that have be assigned to the processors $\mathcal{P}^C$ is $|\mathcal{P}^C| * (c^* - 1)$. Note that $\mathcal{P}^C$ is the set of processors to which the unassigned tasks and the tasks currently assigned to $\mathcal{P}^C$ can be assigned. The total number of tasks is $|\mathcal{P}^C| * (c^* - 1) + |\mathcal{J}^C|$. We have the minimum–maximum load

$$\frac{|\mathcal{P}^C| * (c^* - 1) + |\mathcal{J}^C|}{|\mathcal{P}^C|} > \frac{|\mathcal{P}^C| * (c^* - 1) + |\mathcal{P}^C|}{|\mathcal{P}^C|} = c^*,$$

which contradicts the fact that we can obtain the minimum–maximum load c^* from Algorithm 1. Thus, the assumption $|\mathcal{P}^C| < |\mathcal{J}^C|$ is wrong. So, we have $|\mathcal{P}^C| \geq |\mathcal{J}^C|$. □

Algorithm 4: Match($G, \mathcal{P}^C, \mathcal{J}^C, c^*$)

input : $(G, \mathcal{P}^C, \mathcal{J}^C, c^*)$

output: A matching $G^C = (\{P_i\}, \{\mathcal{J}_i\})$ of the actual maximum-load processors and their tasks

1 **begin**

2 Find the processor nodes $\{P_i\}$ in G that have a load c^* and are in the candidate critical processors $\mathcal{P}^C$ to which unassigned tasks can be assigned;

3 Return the resulting G^C with the critical processors $\{P_i\}$ and the sets $\{\mathcal{J}_i\}$ of tasks that are loaded on them;

4 **end**

We now show a property of the result of the algorithm Match as follows.

Lemma 7. *The algorithm Match finds the actual maximum-load processors $\{P_i\}$, and the number of these processors is no more than the number of unassigned tasks $|\mathcal{J}^C|$ when we set the capacity of the edges (s, P_i) to $c^* - 1$.*

Proof. First, we note that $\{P_i\}$ is defined in the algorithm Match. From Lemma 5, we know that the number of tasks on $\{P_i\}$ is $c^* - 1$ when we set the capacity of the edges (s, P_i) to $c^* - 1$. According to the pigeonhole principle, in order to assign the number $|\mathcal{P}^C| * (c^* - 1) + |\mathcal{J}^C|$ of tasks to the number $|\mathcal{P}^C|$ of processors, there must be no fewer than $|\{P_i\}|$ processors to match the unassigned tasks. So, $\{P_i\}$ can be the set of actual maximum-load processors. $|\{P_i\}| \leq |\mathcal{J}^C|$ follows from the fact that there may be more than one unassigned task that can only be assigned to some processors. □

Finally, we prove that our algorithm ECSEMRPP_Algo (see Algorithm 5) solves the SEMRPP problem optimally for the case of uniform tasks by finding the min–max load vector $\vec{l}$, which is a strongly optimal assignment defined in [21, 23].

Definition 2. *Given an assignment H, we denote by S_k the total load on the*

k most loaded of the processors. We say that an assignment is strongly optimal if, for any other assignment $H^{'}$ ($S_k^{'}$ corresponds accordingly to the total load on the k most loaded of the processors) and for all $1 \leq k \leq m$, we have $S_k \leq S_k^{'}$.

The correctness of ECSEMRPP_Algo is established by the following theorem.

Theorem 1. *The algorithm ECSEMRPP_Algo finds the optimal schedule for the SEMRPP problem in the continuous model in $O(mn^3 \log n)$ time if all tasks have equal numbers of execution cycles.*

Proof. First, we prove that the assignment H returned by ECSEMRPP_Algo is a strongly optimal assignment. We set $H = \{L_1, L_2, \ldots, L_m\}$, where L_i corresponds to the loads of processors P_i in nonascending order. Suppose $H^{'}$ is another assignment such that $H^{'} \neq H$ and $\{L_1^{'}, L_2^{'}, \ldots, L_m^{'}\}$ corresponds to the load. According to the algorithm ECSEMRPP_Algo, we know that $H^{'}$ can only be an assignment in which P_i moves some tasks to P_j $(j < i)$, because P_i cannot move a task to $P_{j'}(j^{'} > i)$, otherwise it could lower the L_i, which is a contradiction to the algorithm ECSEMRPP_Algo. We obtain $\Sigma_{k=1}^{i} L_i \leq \Sigma_{k=1}^{i} L_i^{'}$, i.e., H is a strongly optimal assignment by definition. It turns out that there does not exist any assignment that can reduce the gaps between the loads of the processors in the assignment H. Then the energy consumption of the assignment H' is no less than that of the assignment H as our objective function is convex, so the optimal solution is obtained.

Every time, we discard at least one processor, so the total time cost is $m \times O(n^3 \log n) = O(mn^3 \log n)$ according to Lemma 4, which completes the proof.

Note that in the above analysis, if the load of a processor is less than $s_{\min} \cdot C$, the processor runs at a speed $s_{\min}$. Another prerequisite is that $\max_m \{L_1, \ldots, L_m\} \leq s_{\max} \cdot C$; otherwise there is no feasible solution. □

We use a simple example to illustrate the algorithm.

Example 1. *Suppose there are $m = 3$ processors and $n = 6$ tasks. Given the sets of eligible processors for the tasks $\{\mathcal{M}_j\}$, we construct the network*

Algorithm 5: ECSEMRPP_Algo

input : The set of tasks $\mathcal{J}$, the set of processors $\mathcal{P}$, and the sets of eligible processors for tasks $\{\mathcal{M}_j\}$

output: Scheduling H of tasks on processors

begin
- $G(V, E) = \text{Construct } (\mathcal{J}, \mathcal{P}, \{\mathcal{M}_j\})$;
- Let $G_0(V_0, E_0) = G(V, E)$, $n_0 = n$, $\mathcal{P}^H = \emptyset$, $\mathcal{J}^H = \{\phi_1, \ldots, \phi_m\}$;
- **begin**
  - **while** $G_0 \neq \emptyset$ /**s, t seen as virtual nodes**/ **do**
    - **begin**
      - $(c^*, G_{1st}) = \text{Min\_Max\_Assign}(G_0, n_0)$;
      - $c^* = c^* - 1$;
      - $G_{2ed} = \text{Pre\_Assign}(G_0, c^*)$;
      - $(\mathcal{P}^C, \mathcal{J}^C) = \text{Find\_Candidate\_Critical}($ G_0,G_{2ed},c^*);
      - $G_{3rd} = \text{Match } (G_{1st}, \mathcal{P}^C, \mathcal{J}^C)$;
      - /*According to the scheduling returned by G_{3rd}, we note the processors $\{P_i^H\}$ that have the *actual maximum load* and denote their sets of tasks by $\{\mathcal{J}_i^H\}$. $\mathcal{E}_i^H$ corresponds to the related edges of $\{P_i^H\}$ and $\{\mathcal{J}_i^H\}$*/;
      - $G_0 = \{V_0 \setminus \{P_i^H\} \setminus \{\mathcal{J}_i^H\}, E_0 \setminus \mathcal{E}_i^H\}$;
      - $\mathcal{P}^H = \mathcal{P}^H \cup \{P_i^H\}$, $\phi_i = \mathcal{J}_i^H$, $n_0 = n_0 - \sum_i |\mathcal{J}_i^H|$;
    - **end**
  - **end**
- **end**
- Assign the tasks of $\mathcal{J}_i^H$ to P_i^H and set all the tasks assigned to the processor P_i^H to a speed $(\Sigma_{J_j \in \mathcal{J}_i^H} w_j)/C$. Return the final schedule H.

end

$G = (V, E)$ as shown in Fig. 1(a). In the first iteration, Algorithm 1 finds the minimum-maximum load $c^* = 3$. Suppose the assignment is $\mathcal{J}_1^H = \{J_1, J_2, J_3\}$, $\mathcal{J}_2^H = \{J_4, J_5\}$, $\mathcal{J}_3^H = \{J_6\}$. So, in Algorithm 2, the input is $c_{\text{fix}} = c^* - 1 = 2$ and the maximum flow is 5. Suppose the assignment is $\mathcal{J}_1^H = \{J_1, J_3\}$, $\mathcal{J}_2^H = \{J_4, J_5\}$, $\mathcal{J}_3^H = \{J_6\}$. According to Algorithm 3, the unassigned tasks J_2 can be assigned to processor P_1. Algorithm 4 returns the actual maximum-load of processor P_1 and the set of tasks $\{J_1, J_2, J_3\}$ assigned to it. So, in this iteration, we fix $\mathcal{J}_1^H = \{J_1, J_2, J_3\}$ and delete P_1 and $\{J_1, J_2, J_3\}$, as shown in Fig. 1(b). In the second iteration, according to the same procedure, we can fix $\mathcal{J}_2^H = \{J_4, J_5\}$ and delete P_2 and $\{J_4, J_5\}$, as shown in Fig. 1(c). In the last iteration, we fix $\mathcal{J}_3^H = \{J_6\}$ and delete P_3 and $\{J_6\}$, as shown in Fig. 1(d). After these processes, the algorithm ECSEMRPP_Algo finds the optimal scheduling in which the sets of tasks $\{J_1, J_2, J_3\}$, $\{J_4, J_5\}$, and $\{J_6\}$ are assigned to the processors P_1, P_2, and P_3, respectively.

4.3. General Tasks

As the problem is NP-complete in the strong sense for general tasks (Lemma 2), we aim to obtain an approximation algorithm for the SEMRPP problem. First, we relax the equality (3) of $(\mathbf{P}_1)$ to

$$0 \leq x_{ij} \leq 1 \qquad \forall J_j, P_i \in \mathcal{M}_j. \tag{5}$$

After this relaxation, the SEMRPP problem is transformed to a convex program. The feasibility of this convex program can be checked in polynomial time to within an additive error of ϵ (for an arbitrary constant $\epsilon > 0$) [24], and it can be solved optimally [18]. Suppose x^* is an optimal solution to the relaxed SEMRPP problem. Now our goal is to convert this fractional assignment to an integral one $\bar{x}$. We adopt the method of dependent rounding introduced in [25].

We define a bipartite graph $G(X^*) = (V, E)$, where the vertices of G are $V = \mathcal{J} \cup \mathcal{P}$, and $e = (i, j) \in E$ if $x^*_{ij} > 0$. The weight of edge (i, j) is $x^*_{ij} w_j$. The rounding iteratively modifies x^*_{ij} such that at the end x^*_{ij} becomes integral. There are two main steps, as follows:

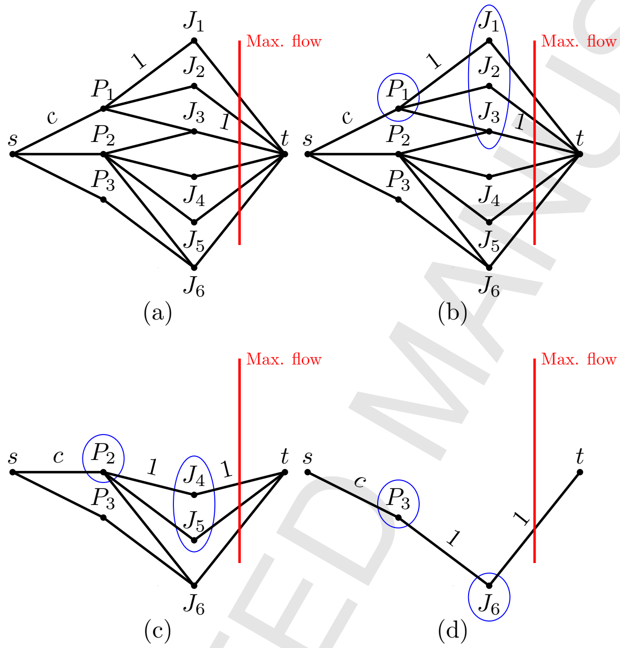

Figure 1: The process of the algorithm ECSEMRPP_Algo. (a) Initial network. (b) First round of finding the maximum load. (c) Second round of finding the maximum load. (d) Third round of finding the maximum load.

1. Break cycle:
   (a) While ($G(x^*)$ has cycle $C = (e_1, e_2, \ldots, e_{2l-1}, e_{2l})$)
      i. Set $C_1 = (e_1, e_3, \ldots, e_{2l-1})$ and $C_2 = (e_2, e_4, \ldots, e_{2l})$;
         Find minimum-weight edge of C, denoted as $e^C_{\min}$, and its weight $\epsilon = \min_{e\in C_1||e\in C_2} e$;
      ii. If $e^C_{\min} \in C_1$, then for every edge in C_1 subtract ϵ and for every edge in C_2 add ϵ;
      iii. Else for every edge in C_1 add ϵ and for every edge in C_2 subtract ϵ;
      iv. Remove the edges with weight 0 from G.
2. Fractional rounding tasks:
   (a) In the first rounding phase, consider each integral assignment if $x^*_{ij} = 1$, set $\bar{x}_{ij} = 1$, and discard the corresponding edge from the graph. Denote the resulting graph by G again;
   (b) While ($G(x^*)$ has connected component C)
      i. Choose one task node from C as root to construct a tree Tr, and match each task node with any one of its children. The resulting matching covers all task nodes;
      ii. Match each task to one of its child nodes (a processor) such that $P_i = \operatorname{argmin}_{P_i\in\mathcal{P}} \Sigma_{\bar{x}_{ij}=1} \bar{x}_{ij} w_j$, set $\bar{x}_{ij} = 1$, and $\bar{x}_{ij} = 0$ for other child nodes.

Lemma 8. *The procedure of relaxation and dependent rounding finds a* 2^α*-approximation to the optimal schedule for the SEMRPP problem in the continuous model in polynomial time.*

Proof. This can be obtained simply from the discussion in [23]. □

We improve this result by analyzing the SEMRPP problem carefully, by generalizing Lemma 8.

Theorem 2. (i) *The procedure of relaxation and dependent rounding finds a* $2^{\alpha-1}(2-1/p^\alpha)$*-approximation to the optimal schedule for the SEMRPP problem* *in the continuous model in polynomial time, where* $p = \max_{\mathcal{M}_j} |\mathcal{M}_j| \leq m$*. (ii) For any processor* P_i*, we have* $\Sigma_{\mathcal{J}} \bar{x}_{ij} w_j < \Sigma_{\mathcal{J}} x^*_{ij} w_j + \max_{\mathcal{J}: x^*_{ij} \in (0,1)} w_j$*, where* x^*_{ij} *is the fractional task assignment at the beginning of the second phase (i.e., the linear constraints on the maximum number of execution cycles are violated only by* $\max_{\mathcal{J}: x^*_{ij} \in (0,1)} w_j$*).*

Proof. (i) Denote the optimal solution to the SEMRPP problem by OPT, denote by H^* the fractional schedule obtained after breaking all cycles, and denote by $\bar{H}$ the schedule returned by the algorithm. Moreover, denote by H_1 the schedule consisting of the tasks assigned in the first step, i.e., $x^*_{ij} = 1$ immediately after breaking the cycles, and denote by H_2 the schedule consisting of the tasks assigned in the second rounding step, i.e., set $\bar{x}_{ij} = 1$ in the matching process. We have $\|H_1\|_\alpha \leq \|H^*\|_\alpha \leq \|OPT\|_\alpha$,[2] where the first inequality follows from the fact that H_1 is a subschedule of H^* and the second inequality results from H^* being a fractional optimal schedule compared with OPT, which is an integral schedule. We consider $\|H_1\|_\alpha \leq \|H^*\|_\alpha$ carefully. If $\|H_1\|_\alpha = \|H^*\|_\alpha$, that is, all tasks have been assigned in the first step and the second rounding step is not necessary, then we have $\|H_1\|_\alpha = \|H^*\|_\alpha = \|OPT\|_\alpha$, such that the approximation is 1. Next we consider $\|H_1\|_\alpha < \|H^*\|_\alpha$, so that there are some tasks assigned in the second rounding step; without loss of generality, we denote these as $\mathcal{J}_1 = \{J_1, \ldots, J_k\}$. We assume that the fraction of task J_j assigned to processor P_i is f_{ij} and the largest size of the eligible processor set

[2] In the schedule H_1, when the loads of m processors are $\{l^{h1}_1, l^{h1}_2, \ldots, l^{h1}_m\}$, $\|H_1\|_\alpha$ means $((l^{h1}_1)^\alpha + (l^{h1}_2)^\alpha + \ldots + (l^{h1}_m)^\alpha)^{1/\alpha}$.

is $p = \max_{\mathcal{M}_j} |\mathcal{M}_j| \le m$. Then we have

$$
\begin{aligned}
(\|H^*\|_\alpha)^\alpha &= \sum_{i=1}^{m} (\Sigma_{J_j : x^*_{ij}=1} w_j + \Sigma_{J_j \in \mathcal{J}_1} f_{ij})^\alpha \\
&\ge \sum_{i=1}^{m} (\Sigma_{J_j : x^*_{ij}=1} w_j)^\alpha + \sum_{i=1}^{m} (\Sigma_{J_j \in \mathcal{J}_1} f_{ij})^\alpha \\
&= (\|H_1\|_\alpha)^\alpha + \sum_{i=1}^{m} (\Sigma_{J_j \in \mathcal{J}_1} f_{ij})^\alpha \\
&\ge (\|H_1\|_\alpha)^\alpha + \sum_{i=1}^{m} \sum_{j=1}^{k} (f_{ij})^\alpha \qquad (6) \\
&= (\|H_1\|_\alpha)^\alpha + \sum_{j=1}^{k} \sum_{i=1}^{m} (f_{ij})^\alpha \\
&\ge (\|H_1\|_\alpha)^\alpha + \sum_{j=1}^{k} \left(\frac{\sum_{i=1}^{m} f_{ij}}{p} \right)^\alpha \\
&= (\|H_1\|_\alpha)^\alpha + \frac{1}{p^\alpha} \sum_{j=1}^{k} (w_j)^\alpha,
\end{aligned}
$$

where f_{ij} is the fraction of task J_j assigned to processor P_i. From the fact that H_2 schedules only one task per processor, this is the optimal integral assignment for the subset of tasks that it assigns, and it certainly has a cost smaller than any integral assignment for the whole set of tasks. In a similar way, we have

$$(\|H_2\|_\alpha)^\alpha = \sum_{j=1}^{k} (w_j)^\alpha \le (\|OPT\|_\alpha)^\alpha. \qquad (7)$$

So, the inequality (6) can be reduced to

$$(\|H^*\|_\alpha)^\alpha \ge (\|H_1\|_\alpha)^\alpha + \frac{1}{p^\alpha} (\|H_2\|_\alpha)^\alpha, \qquad (8)$$

and then

$$\begin{aligned}
(\|\bar{H}\|_\alpha)^\alpha &= (\|H_1 + H_2\|_\alpha)^\alpha \leq (\|H_1\|_\alpha + \|H_2\|_\alpha)^\alpha \\
&= 2^\alpha \left(\frac{\|H_1\|_\alpha + \|H_2\|_\alpha}{2} \right)^\alpha \\
&\leq 2^\alpha \left(\frac{1}{2}(\|H_1\|_\alpha)^\alpha + \frac{1}{2}(\|H_2\|_\alpha)^\alpha \right) \\
&\leq 2^{\alpha-1}((\|H^*\|_\alpha)^\alpha - \frac{1}{p^\alpha}(\|H_2\|_\alpha)^\alpha + (\|H_2\|_\alpha)^\alpha) \\
&\leq 2^{\alpha-1} \left(2 - \frac{1}{p^\alpha} \right) (\|OPT\|_\alpha)^\alpha.
\end{aligned}$$

So,

$$\frac{(\|\bar{H}\|_\alpha)^\alpha}{(\|OPT\|_\alpha)^\alpha} \leq 2^{\alpha-1} \left(2 - \frac{1}{p^\alpha} \right).$$

This concludes the proof that the schedule $\bar{H}$ guarantees a $2^{\alpha-1}(2 - 1/p^\alpha)$-approximation to the optimal solution for the SEMRPP problem and can be found in polynomial time.

(ii) From the above, we also have

$$\Sigma_{J_j \in \mathcal{J}} \bar{x}_{ij} w_j < \Sigma_{J_j \in \mathcal{J}} x^*_{ij} w_j + \max_{J_j \in \mathcal{J}: x^*_{ij} \in (0,1)} w_j, \forall P_i,$$

where the inequality results from the fact that the load of processor P_i in schedule $\bar{H}$ is the load of H^* plus the weight of the tasks matched to it. Because we match each task to one of its child nodes, we find that the number of execution cycles of the added task satisfies the inequality $\bar{w}_j < \max_{J_j \in \mathcal{J}: x^*_{ij} \in (0,1)} w_j$. □

Now we discuss $s_{\max}$. First, we present a claim about the relationship between feasibility and violation.

Claim 1. *If* $(\mathbf{P}_1)$ *(the SEMRPP problem in the continuous model) has a feasible solution, it is hard to solve* $(\mathbf{P}_1)$ *without violating the constraint of the limitation on the maximum number of execution cycles of the processors.*

Obviously, if $(\mathbf{P}_1)$ has a unique feasible solution, the maximum number of execution cycles of the processors is set to the value given by the solution OPT. Then, if we can always solve $(\mathbf{P}_1)$ without violating the constraint, this means that we can easily devise an exact algorithm for $(\mathbf{P}_1)$. But we have proof that $(\mathbf{P}_1)$ is NP-complete in the strong sense.

Next, we give a guarantee speed which can be regarded as fast enough in the procedure of dependent rounding.

435 **Lemma 9.** *The procedure of dependent rounding can provide an approximate solution without violating the constraint on the maximum number of execution cycles of the processors when* $s_{\max}C \geq \max_{P_i \in \mathcal{P}} L_i + \max_{J_j \in \mathcal{J}} w_j$*, where* $L_i = \Sigma_{J_j \in \mathcal{J}_i}(1/|\mathcal{M}_j|)w_j$*, and* $\mathcal{J}_i$ *is the set of tasks that can be assigned to processor* P_i.

Proof. First, we define a vector $\vec{H} = \{H_1, H_2, \ldots, H_m\}$, in nonascending sorted order, as the numbers of execution cycles of m processors at the beginning of the second step. We also define a vector $\vec{L} = \{L_1, L_2, \ldots, L_m\}$, in nonascending sorted order, as the numbers of execution cycles of m processors such that $L_i = \Sigma_{J_j \in \mathcal{J}_i}(1/|\mathcal{M}_j|)w_j$. Now we need to prove that $H_1 \leq L_1$. Suppose we have $H_1 > L_1$; without loss of generality, we assume that the processor P_1 has a number of execution cycles H_1. We denote the set of tasks assigned to P_1 by $\mathcal{J}_1^H$. Let $\mathcal{M}_1^H$ be the set of processors to which a task, currently fractionally or integrally assigned to processor P_1, can be assigned, i.e., $\mathcal{M}_1^H = \bigcup_{J_j \in \mathcal{J}_1^H} \mathcal{M}_j$. Similarly, we denote the set of tasks that can be processed on $\mathcal{M}_1^H$ by $\mathcal{J}^H$, and the set of processors $\mathcal{M}^H$ for every task in $P_i \in \mathcal{M}_1^H$ can be assigned. We have $\mathcal{M}^H = \bigcup_{J_j \in \mathcal{J}^H} \mathcal{M}_j$. Without loss of generality, we define $\mathcal{M}^H$ as a set $\{h_1, h_2, \ldots, h_k\}$ $(1 \leq k \leq m)$ and also define a set $\{l_1, l_2, \ldots, l_k\}$ $(1 \leq k \leq m)$ as its respective set of processors in $\vec{L}$. According to the convexity of the objective function, we obtain $H_{h_1} = H_{h_2} = \ldots = H_{h_k}$. By our assumption, $H_{h_p} > L_{l_q}$, $\forall p, \forall q$. Then

$$\Sigma_p H_{h_p} > \Sigma_q L_{l_q}. \tag{9}$$

Note that each integral task (at the beginning of the second step) in the left part of inequality (9) can also have its respective integral task in the right part, but the right part may contain some fractional tasks. So, $\Sigma_q L_{l_q} - \Sigma_p H_{h_p} \geq 0$, i.e., $\Sigma_p H_{h_p} \leq \Sigma_q L_{l_q}$, a contradiction to inequality (9). The assumption is wrong;

we have $H_1 \leq L_1$. By Theorem 2, where the maximum number of execution cycles for the dependent rounding is $\bar{H}_{\max}$, we have

$$\begin{aligned}\bar{H}_{\max} &< H_1 + \max\nolimits_{J_j \in \mathcal{J}: x^*_{ij} \in (0,1)} w_j \\ &\leq L_1 + \max\nolimits_{J_j \in \mathcal{J}: x^*_{ij} \in (0,1)} w_j \\ &\leq L_1 + \max\nolimits_{J_j \in \mathcal{J}} w_j = \max\nolimits_i L_i + \max\nolimits_{J_j \in \mathcal{J}} w_j.\end{aligned}$$

This completes the proof. □

5. Numerical Results

In this section, we provide performance details based on numerical results. To demonstrate the effectiveness of our approach, we compared five values of interest, namely the optimal fractional solution, the optimal integral solution, the fractional-dependent-rounding (FDR) integral solution (in the rest of the paper, this refers to the solution obtained by our algorithm), the least-flexible-task (least flexible job, LFJ) solution, and the least-flexible-processor (least flexible machine, LFM) solution. We used the CPLEX solver [26] to obtain the optimal integral solution by solving the relevant integer programming problem. For our approximation algorithm, we obtained the optimal fractional solution by use of the CVX solver [27], and then applied dependent rounding by means of our algorithm. The LFJ and LFM solutions were obtained by the following LFJ and LFM algorithms:

- *LFJ algorithm.* The tasks are first sorted in nondecreasing order of the cardinality of their eligible processing sets, i.e., by $|\mathcal{M}_j|$. All the tasks are then scheduled in this order by sequential list. Next, each task is assigned to a processor P_i which has the least load and is in that task's eligible processing set ($P_i \in \mathcal{M}_j$). Finally, the speed of each processor is set to a value such that the processor finishes its load in accordance with the time constraint.

- *LFM algorithm.* The processors are first sorted in nondecreasing order of the cardinality of their sets of eligible processing tasks. The processors are then scheduled in this order by sequential list. Next, each processor chooses a task which can be assigned to it and has not been assigned to another processor. Finally, the speed of each processor is set to a value such that the processor finishes its load in accordance with the time constraint. Note that the main difference between the LFJ and LFM algorithms is between whether tasks or processors are the objects used to select the other party (processors or tasks, respectively).

5.1. Simulation Setting

To evaluate the performance of our algorithm, we created systems consisting of 10–50 processors and 50–300 tasks. Each task J_j was characterized by two parameters: the number of execution cycles w_j and the eligible processing set $\mathcal{M}_j$; w_j was randomly generated in the range $[1, 10\,000]$. We simulated two cases for $\mathcal{M}_j$: one was randomly generated from the set $\mathcal{P}$ of processors, and the other was arranged to conform to the restriction of inclusive processing sets [11].[3] The maximum speed $s_{\max}$ was set to a value large enough to allow a feasible solution to be obtained. We analyzed four different cases, where we varied the tightness of the time constraint C, used two different power parameters α, varied the ratio η of the number of tasks to the number of processors, and used two different eligible processing sets. Without loss of generality, we set the power parameter α to 2 when studying the other cases. All of the results presented are mean values from several different runs on an Intel Core I5-2400 CPU with a speed of 3.10 GHz $\times$ 4.

5.2. Simulation Results

Figure 2(a) represents the energy consumption of a system with 10 processors and 27 tasks when the time constraint is increased. The five curves correspond to the values of the five solutions mentioned at the beginning of this section.

[3] "Inclusive processing sets" means that for a pair of restricted processing sets $\mathcal{M}_j$ and $\mathcal{M}_k$ for any two different tasks, either $\mathcal{M}_j \subseteq \mathcal{M}_k$ or $\mathcal{M}_k \subseteq \mathcal{M}_j$.

Figure 2(b) reports the relative energy consumption ratios for these five values, where all of them have been normalized by the optimal integral solution. Some observations from this simulation are as follows. (1) As shown in Figure 2(a) and (b), the energy consumption and the time constraint are in inverse proportion, and the ratios are almost not influenced by the different time constraints. This confirms Lemma 1 and Corollary 1, i.e., each processor executes all tasks that are assigned to it at a uniform speed. So, when the time constraint C is increased to $k \times C$, each processor can lower its speed to s/k to finish the tasks. For $\alpha = 2$, the energy consumption is equal to $(1/k)$ times the energy consumption when the time constraint is not increased (i.e., $k \times C \times (s/k)^2 = (1/k)(Cs^2)$). Thus each kind of energy consumption is influenced in the same proportion by variation of the time constraint; when the energy consumption is normalized by the optimal integral solution, the time constraint can be removed. (2) The optimal fractional solutions are little different from the integral optimal solutions: the gap is within 5% in the simulations. A similar difference can also be observed between the integral optimal solution and the fractional-dependent-rounding integral solution; in fact, this difference is also about 5% in the simulations. This suggests that the FDR solution performs much better than the approximation ratio that we analyzed in Theorem 2. (3) The figure confirms the superiority of the fractional-dependent-rounding integral solution, as this solution can reach values 11% and 15% better than the LFJ and LFM solutions, respectively. After checking the load of the processor with the maximum load, we found that the fractional-dependent-rounding solution was close to the integral optimal solution. This suggests that the fractional-dependent-rounding integral solution can balance the load between the eligible processing sets more efficiently.

Figure 3 illustrates the normalized energy consumption ratios for two different power parameters. As can be seen from the figure, when the power parameter α increases, our solution becomes more competitive. More precisely, the saving in energy consumption changes from 11% at $\alpha = 2$ to 31% at $\alpha = 3$ compared with the LFJ solution, and from 15% at $\alpha = 2$ to 40% at $\alpha = 3$ compared with the LFM solution. This is because a larger power parameter

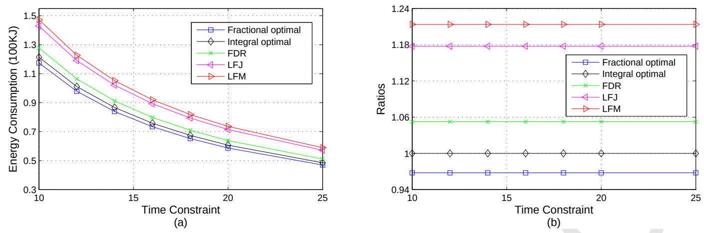

Figure 2: Performance of five solutions with different time constraints. (a) Energy consumption. (b) Energy consumption ratio normalized by the optimal integral solution.

amplifies an improper allocation between processors and tasks.

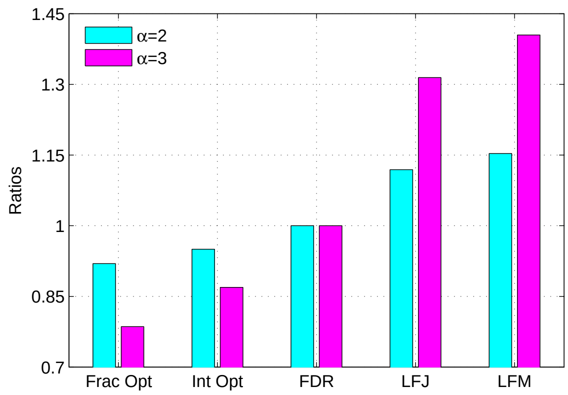

Figure 3: Energy consumption ratio normalized by the FDR solution for two power parameters α. (Frac Opt = fractional optimal; Int Opt = integral optimal.)

Figure 4 depicts the normalized energy consumption ratios for different solutions with a varying ratio η of the number of tasks to the number of processors. When the ratio η is small, the difference between the normalized ratios is much larger. This can be explained by the fact that if only one task was improperly assigned, the energy consumption would oscillate excessively if η was small. As η is increased, the oscillation will be reduced because an improper task assignment will not influence the result very much.

Figure 5 illustrates the normalized energy consumption ratios of a system

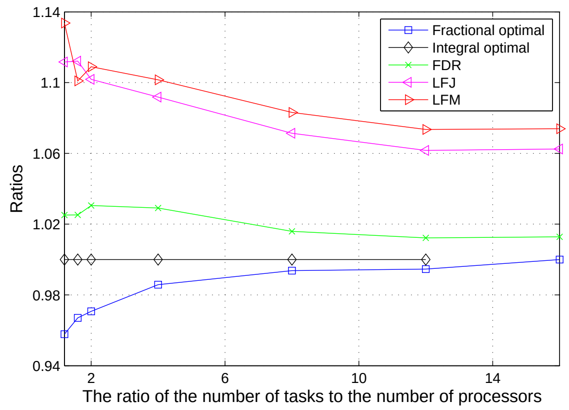

Figure 4: Energy consumption ratio normalized by the optimal integral solution for varying ratio η. (The value of the optimal integral solution is missing for the last point because it could not be obtained, owing to problems with both memory and running time. The last points for the other values are normalized by the optimal fractional solution.)

with 14 processors and 35 tasks for two eligible processing sets. As shown in the figure, the different eligible processor sets can influence the performance of the algorithms. The FDR and LFJ solutions perform better in the case of a random processor set. This can be explained by the fact that in the LFJ solution and the FDR solution (in the last stage, when fractional tasks for processors are rounded) the tasks choose their processors, and the random restrictions help the tasks make the proper choice, but the difference is not so obvious. In contrast, the LFM solution, in which the processors choose their tasks, performs much better in the case of inclusive processing sets. This can be explained by the fact that the processor which has the smallest number of eligible tasks selects a task first; if it makes an improper choice, the subsequent processors will not be influenced much, as they have more tasks to choose from in the case of inclusive processing sets. It is interesting to observe that the algorithms perform very differently under random and regular conditions.

The average running times for the optimal fractional solution obtained by CVX, the fractional-dependent-rounding integral solution obtained by CVX and

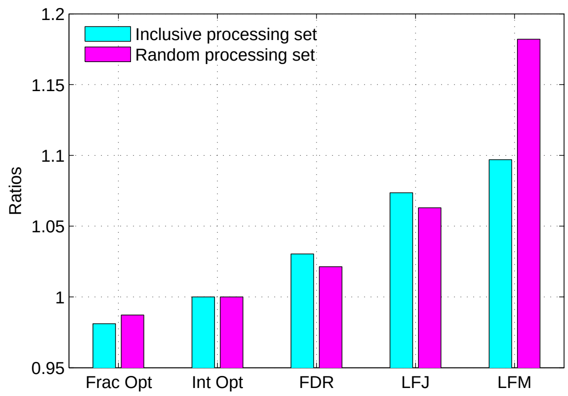

Figure 5: Energy consumption ratio normalized by the optimal integral solution for two eligible processing sets. (Frac Opt = fractional optimal; Int Opt = integral optimal.)

rounding, the LFJ solution obtained by the LFJ algorithm, and the LFM solution obtained by the LFM algorithm were short (in our simulations, they took at most several minutes) in all the instances presented here. However, obtaining the optimal integral solution by CPLEX took more than one day for large systems. For larger systems, the optimal integral solution had problems with both memory and running time. Note that in all of the simulations, the FDR solution was more efficient than the LFJ and LFM solutions. This suggests that our solution could assign tasks more appropriately in every instance, and be able to solve the SEMRPP problem efficiently owing to the high quality of the solutions and low computational time.

6. Conclusion

In this paper, we have explored algorithmic instruments aimed at reducing energy consumption with restricted parallel processors. We aimed to minimize the total energy consumption, and the speed scaling method was used to save energy under an execution time constraint. We first assessed the complexity of the scheduling problem given a time constraint and the setting of restricted parallel processors. Specially, for the case where the tasks have a uniform size, we have proposed an optimal scheduling algorithm with time complexity $O(mn^3 \log n)$. We then presented a polynomial-time approximation algorithm with an approximation factor $2^{\alpha-1}(2 - 1/p^{\alpha})$ (where $p = \max_{\mathcal{M}_j} |\mathcal{M}_j|$) for the general case in which the tasks have an arbitrary size measured in execution cycles. We evaluated the performance of the approximation algorithm by a set of simulations after analysis of the algorithm. It turns out that our solution is closer than other solutions to the optimal solution. This confirms that our algorithm could provide more efficient scheduling for the SEMRPP problem.

Acknowledgment

The authors would like to thank anonymous reviewers for their helpful comments, which have greatly improved the manuscript. This research was partially supported by NSFC Major International Collaboration Project 61020106002, NSFC & Hong Kong RGC Joint Project 61161160566, NSFC Project for Innovation Groups 61221062, NSFC Project grant 61202059, the Comunidad de Madrid grant S2009TIC-1692, and Spanish MICINN/MINECO grant TEC2011-29688-C02-01.

References

[1] T. Mudge. Power: A first-class architecture design constraint. *Computer, 34(4)*, pages 52–58, 2001.

[2] G. Aupy, A. Benoit, F. Dufossé, and Y. Robert. Reclaiming the energy of a schedule: Models and algorithms. *Concurrency and Computation: Practice and Experience, 25(11)*, pages 1505–1523, 2013.

[3] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 374–382, 1995.

[4] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'98)*, pages 197–202, 1998.

[5] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 37–46, 2003.

[6] J. Chen and W. Kuo. Multiprocessor energy-efficient scheduling for real-time jobs with different power characteristics. In *International Conference on Parallel Processing (ICPP'05)*, pages 13–20, 2005.

[7] S. Albers, F. Müller, and S. Schmelzer. Speed scaling on parallel processors. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, pages 289–298, 2007.

[8] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems, 43(1)*, pages 67–80, 2008.

[9] G. Greiner, T. Nonner, and A. Souza. The bell is ringing in speed-scaled multiprocessor scheduling. In *Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'09)*, pages 11–18, 2009.

[10] E. Angel, E. Bampis, F. Kacem, and D. Letsios. Speed scaling on parallel processors with migration. In *Proceedings of the 18th International Conference on Parallel Processing (EuroPar'12)*, pages 128–140, 2012.

[11] J. Leung and L. Li. Scheduling with processing set restrictions: A survey. *International Journal of Production Economics, 116(2)*, pages 251–262, 2008.

[12] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the Conference on Power Aware Computing and Systems (HotPower'08)*, 2008.

[13] A. Gupta, S. Im, R. Krishnaswamy, B. Moseley, and K. Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *Proceedings of the 23rd Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'12)*, pages 1242–1253, 2012.

[14] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang. A throughput optimal algorithm for map task scheduling in mapreduce with data locality. In *ACM Sigmetrics Performance Evaluation Review, 40(4)*, pages 33–42, 2013.

[15] U.S.Environmental Protection Agency. Server energy and efficiency report. 2009.

[16] B. Barrett. Google's insane number of servers visualized. http://www.gizmodo.com/5517041.

[17] J. L. W. V. Jensen. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Mathematica, 30(1)*, pages 175–193, 1906.

[18] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[20] J. Leung. *Handbook of Scheduling*. CRC Press, Boca Raton, FL, 2004.

[21] N. Alon, Y. Azar, G. Woeginger, and T. Yadid. Approximation schemes for scheduling. In *Proceedings of the 8th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 493–500, 1997.

[22] Y. Lin and W. Li. Parallel machine scheduling of machine-dependent jobs with unit-length. *European Journal of Operational Research, 156(1)*, pages 261–266, 2004.

[23] Y. Azar, L. Epstein, Y. Richter, and G. Woeginger. All-norm approximation algorithms. *Journal of Algorithms, 52(2)*, pages 120–133, 2004.

[24] Y. Nesterov and A. Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*, SIAM Studies in Applied Mathematics. SIAM, 1994.

[25] R. Gandhi, S. Khuller, S. Parthasarathy, and A. Srinivasan. Dependent rounding in bipartite graphs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'02)*, pages 323–332, 2002.

[26] IBM. CPLEX Optimizer. http://www.ilog.com/products/cplex.

[27] CVX Research. CVX: Matlab software for disciplined convex programming. http://www.cvxr.com/cvx.

Xibo Jin is a PhD candidate of Institute of Computing Technology, Chinese Academy of Sciences, and a student member of IEEE. His research interests include parallel computing, Cloud computing, scheduling algorithms, and energy-efficient computing.

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

Liya Fan has a PhD from the Institute of Computing Technology, Chinese Academy of Sciences. Now he is a staff researcher of IBM China Research Laboratory. His research interests include parallel computing and scheduling algorithms.

Ying Song received the PhD degree in computer engineering from the Institute of Computing Technology, Chinese Academy of Sciences. She is an assistant professor at the State Key Laboratory of Computer Architecture at the Institute of Computing Technology. Her main research interests include computer architecture, parallel and distributed computing, and virtualization technology.

Zhiyong Liu is a professor of the Institute of Computing Technology, Chinese Academy of Sciences, and a senior member of China Computer Federation. His research interests include high performance algorithms and architecture, and parallel processing.

HIGHLIGHTS:

We propose an optimal scheduling algorithm for the case when all of the tasks have uniform computational work.
We present a polynomial-time algorithm that achieves a bounded approximation factor when the tasks have arbitrary-size work.
We evaluate the performance of the approximation algorithm by a set of simulations.