Scheduling Mixed Tasks with Deadlines in Grids Using Bin Packing

Cong Liu\(^1\) and Sanjeev Baskiyar\(^2\)
\(^1\)Department of Computer Science, The University of North Carolina at Chapel Hill
\(^2\)Department of Computer Science and Software Engineering, Auburn University
cong@cs.unc.edu

Abstract

This paper addresses the problem of scheduling independent tasks with different priorities and deadline constraints in computational grids. The problem is first formulated as a bin packing problem. We propose a heuristic algorithm named Residual Capacity Maximization Scheduling (RCMS), which integrates the ideas of a classical bin packing algorithm (Best Fit) and a mixed integer quadratic programming modeling approach. RCMS is highly scalable as it does not need to know the global state of the grid. Simulation results based on a real-world trace demonstrate that with respect to the total number of schedulable tasks meeting deadlines, RCMS outperforms existing approaches by 8%. With respect to the number of the highest priority tasks meeting deadlines, RCMS outperforms them by over 20% on average. Moreover, RCMS is a fully distributed algorithm with a low complexity.

1. Introduction

Grid computing [3] consists of the coordinated use of heterogeneous resources with different capabilities to satisfy varying computation requirements specified by users. Many users have Quality of Service (QoS) requirements such as deadline and priority. Application scheduling algorithms must consider such QoS requirements. Handling mixed task priorities can help open the grid and make it amenable for commercialization. In this paper, we consider scheduling a bag of independent tasks with mixed types (i.e. priorities) and deadlines in grids. Three types of tasks are considered: hard, firm and soft [5]. It is important for a grid scheduler to provide performance guarantees to hard tasks while not degrading performance for other types of tasks.

To the best of our knowledge, this paper is the first attempt to model such a problem into a formalized bin packing problem. RCMS prioritizes tasks according to task types as well as deadline. Motivated by the Best Fit algorithm, RCMS proposes a mixed integer quadratic programming model that always maximizes the residual capacity of resources at each step following a task-resource mapping. Doing so may create more and larger integrated slacks, thus giving firm and soft tasks higher probabilities to be scheduled into such slacks. RCMS contains a peer-to-peer scalable dispatching strategy, thus being highly scalable as it only requires partial knowledge of its peers, not a full knowledge of the state of all grid sites.

The remainder of this paper is organized as follows. A review of recent related works is given in Section 2. In Section 3, the problem formulation is described. Section 4 presents the detailed design of RCMS. Section 5 presents a comprehensive set of simulations that evaluate the performance of RCMS. Finally, conclusions and suggestions for future work appear in Section 6.

2. Related work

Several effective scheduling algorithms such as Sufferage [7], Min-Min [8], have been proposed in previous works. The rationale behind Sufferage is to allocate a site to a task that would “suffer” most in completion time if it were not allocated to that site. For each task, its sufferage value is defined as the difference between its best minimum completion time and its second-best minimum completion time. The complexity of the Sufferage algorithm is \(O(msn^2)\) if applied in a grid system, where \(m\) is the total number of machines, \(s\) is the number of sites within the grid, and \(n\) is the number of incoming tasks. The Min-Min heuristic begins with computing the set of minimum completion time and its second-best minimum completion time. The complexity of the Sufferage algorithm is \(O(msn^2)\) if applied in a grid system.
Little research exists on scheduling algorithms considering both task types and deadlines in grids. A deadline based scheduling algorithm appears in [12] for multi-client and multi-server within a single cluster. The algorithm aims at minimizing deadline misses by using load correction and fallback mechanisms. Its complexity is \(O(msn)\) if applied in a grid system. Based on this work, in [2], a preemptive scheduling algorithm with deadline and priority concern appropriate for multi-client, multi-server within a single cluster has been proposed. Since preemption is allowed, it leaves open the possibility that tasks with lower priority but early deadlines may miss deadlines. Furthermore, the authors do not present an estimate of the fraction of successfully schedulable high and low priority tasks. Its complexity is \(O(nm^2s^2)\), if applied to a grid system.

In [4], a number of classical heuristics (e.g. Max-Min, Max-Max, Percentage Best [8], Relative Cost [14]) are revised to map tasks with priorities and deadlines in a single cluster environment. However, these revised algorithms do not provide guarantee to complete hard tasks before deadlines. Moreover, since the target hardware platform is a single cluster, they have not taken the data transfer delays into consideration. Also scalability has not been considered as they deal with a single cluster.

The aforementioned algorithms do not consider all of the following criteria: task types, dispatch times, deadlines, scalability and distributed scheduling. Furthermore, they require a full knowledge of the state of the grid which is difficult and/or expensive to maintain. Recently, we introduced the problem of scheduling mixed tasks in grids, and developed a General Distributed Scheduler (GDS) to solve this problem [6]. In this work, we first rank tasks and schedule each task to its latest possible start time at the local resource. Next, we use a p2p dispatching strategy to execute unscheduled tasks in remote sites. The present work proposes a quadratic programming model that maximizes the slacks in a formalized way.

3. Problem statement

This paper targets a grid system which consists of many geographically distributed sites interconnected through WAN. We define a site as a location that contains many computing resources. Heterogeneity and dynamicity cause resources in a grid to be distributed hierarchically or in clusters rather than uniformly. Within the same site, we assume that the computing capacities of resources are almost identical or the difference between any two resources is so small that it can be ignored. At each site, there is a main server and several supplemental servers, which collect information from all machines within that site. If the main server fails, a supplemental server will take over. Intra-site communication cost is usually negligible as compared to inter-site communication.

We consider scheduling Bag-of-Tasks (BoT) applications, which are those parallel applications whose tasks are independent of one another. Because of the independence of their tasks, BoT applications can be successfully executed over geographically distributed computational grids, as has been demonstrated by SETI@home [1]. Three types of tasks are considered: hard, firm and soft. RCMS uses such a task taxonomy which considers the consequence of missing deadlines, and the importance of task property. Hard tasks have the highest priority in that the consequences of failure are catastrophic, e.g. computing the orbit of a moving satellite to make real-time defending decisions [9]. For firm tasks a few missed deadlines will not lead to total failure, but such a failure may occur when more deadlines are missed. An example of a firm task with deadline is of the George E. Brown, Jr. Network for Earthquake Engineering Simulation (NEES) [10]. For soft tasks, failures will only result in degraded performance. Many applications which fall in the category of soft tasks include coarse-grained task-parallel computations arising from parameter sweeps, Monte Carlo simulations, and data parallelism.

A set of \(n\) independent tasks with different types and deadlines are submitted at each site. The goal is to guarantee that all hard tasks can be scheduled within their deadlines, while maximizing the number of firm and soft tasks that are successfully scheduled within deadlines.

3.1. Bin-packing model

We modeled the aforementioned scheduling problem as a variation of the standard bin packing problem. Given a set of \(n\) tasks (items) and a set of \(m\) processors (bins) at a site, we find a schedule that packs the maximum number of tasks into processors.

We are given a list \(L = (a_1, \ldots, a_n)\) of items, with variable item size \(size(a_i)\), deadline \(d_i\), and priority \(p_i\), submitted to \(m\) bins \((b_1, \ldots, b_m)\) with the same size \(size_b\), since resources within the same site are assumed to be homogeneous. The value of \(size_b\) is decided by:

\[
size_b = \text{Max} \{d_1, \ldots, d_n\}. \tag{1}
\]

In other words, it equals to the longest deadline among all items. Therefore, each bin is large enough to accommodate any single item or any single deadline when no item is assigned, given that an item is dropped
if it cannot be packed within the deadline. A deadline value represents the urgency of an item, while a priority value means its importance. Each bin $b_i$ has a neighbor list, indicating that bins in this list have a shorter distance to itself than other bins. Transferring an item $a_j$ from $b_j$ to any other bin $b_k$ requires a certain amount of time $t_{ijk}$, depending on the distance from $b_j$ to $b_k$. Each item size varies on different bins. For an item $a_j$ from $b_j$, we define a valid bin bound $v_i = (r_{ib}, \delta_j)$, where $r_{ik}$ is the arrival time of item $a_i$ at bin $b_k$.

$$
\begin{align*}
    r_{ik} &= 0 & \text{if } j = k \\
    r_{ik} &= t_{ijk} & \text{if } j \neq k.
\end{align*}
$$

Only if $a_k$ is packed within bound $v_k$, can it be considered a valid pack. To summarize, the grid scheduling problem modeled as a variant of the bin-packing problem is defined in the following way:

- The items and their variable sizes on different bins are given initially, and all bins are ready at the very beginning.
- Each item must be packed in its valid bin bound. Each item’s arrival times at different bins are known initially.
- Preference must be given to the items with higher priority.
- The goal is to pack as many given items as possible into the bins and guarantee that all high priority items are packed successfully.

This problem is a combinatorial NP-hard problem; the worst-case computational complexity of exactly solving this problem is expected to be exponential. We propose a heuristic approach, RCMS, based on the classical Best Fit of the Bin Packing problem. The core idea of the Best Fit algorithm is that it always picks the bin with the least amount of free space that can hold the current item. A detailed motivational example behind using the Best Fit idea is given in Section 4.3.

4. RCMS algorithm

The RCMS algorithm is designed in order to schedule all hard tasks while maximizing total number of firm/soft tasks that can meet deadlines. It consists of the following sub-procedures: Schedule, Pack, and Dispatch.

First incoming tasks at each site are ranked based upon priority and deadline. Next, a quadratic programming model based scheduling algorithm is used to assign each task to a specific resource on a site. Then, the schedule is packed to increase fault tolerance. Finally those tasks that are unable to be scheduled are dispatched to remote sites where the

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_i$</td>
<td>task $i$</td>
</tr>
<tr>
<td>$e_{ij}$</td>
<td>estimated execution time of $t_i$ on machine $m_i$ at site $s_i$</td>
</tr>
<tr>
<td>$c_d$</td>
<td>estimated transmission time of $t_i$ from current site to $s_j$</td>
</tr>
<tr>
<td>$s_{ik}$</td>
<td>start time of tasks $t_i$ on $m_i$ at $s_j$</td>
</tr>
<tr>
<td>$e_i$</td>
<td>instruction size of $t_i$, in terms of MI (Million Instruction)</td>
</tr>
<tr>
<td>$CC_i$</td>
<td>computing capacity of $s_i$</td>
</tr>
<tr>
<td>$BW_{ij}$</td>
<td>network bandwidth between $s_i$ and $s_j$</td>
</tr>
<tr>
<td>$Ave_{CC_i}$</td>
<td>average computing capacity of all the neighboring sites of $s_i$</td>
</tr>
<tr>
<td>$Ave_{T_{ij}}$</td>
<td>estimated average transmission time of $t_i$ from $s_j$ to all the neighbors</td>
</tr>
<tr>
<td>$d_i$</td>
<td>deadline of $t_i$</td>
</tr>
<tr>
<td>$CCR_{ij}$</td>
<td>communication to computation ratio of $t_i$ residing at $s_j$</td>
</tr>
<tr>
<td>$cc_{ij}$</td>
<td>computing capacity of $m_i$ at $s_j$, in terms of MIPS (Million Instruction PerSecond)</td>
</tr>
<tr>
<td>$m_i$</td>
<td>number of machines within $s_i$</td>
</tr>
<tr>
<td>$p_{ij}$</td>
<td>pre-bound of $p_i$’ task slot on $m_i$ at $s_j$</td>
</tr>
<tr>
<td>$st_{ij}$</td>
<td>start time of $p_i$’ task slot on $m_i$ at $s_j$</td>
</tr>
<tr>
<td>$ed_{ij}$</td>
<td>end time of $p_i$’ task slot on $m_i$ at $s_j$</td>
</tr>
<tr>
<td>$sl_{ij}$</td>
<td>length of $p_i$’ slack on $m_i$ at $s_j$</td>
</tr>
<tr>
<td>$rc_{ij}$</td>
<td>residual capacity of $m_i$ at $s_j$</td>
</tr>
</tbody>
</table>

We define a data structure called task slot. A task slot contains two parts: (i) slack time and (ii) task execution period. The slack time of a task slot is defined as the available time period between this task slot’s pre-bound and its start time. The pre-bound of the task slot equals to its previous task slot’s end time, if one existed. For the first task slot, its pre-bound is 0.

4.1. Notations

Table 1 shows the notations used in this paper.

![Fig. 1. RCMS main function](image-url)
The start/end time of a task slot is the same start/end time of the corresponding task. Fig. 2 shows the concept of a task slot.

Fig. 2. Illustration of the task slot data structure

A task is composed of execution code, input and output data, priority, deadline, and CCR-type. Tasks are assigned one of the following priorities: high, normal, or low, which correspond to hard, firm, and soft type. Task $t_i$’s CCR-type, denoted as $CCR_{ij}$, is decided by its Communication to Computation Ratio (CCR), which represents the relationship between the transmission time and execution time of a task. It can be defined as:

$$CCR_{ij} = \frac{\text{Ave}_T_{ij}}{\text{Ave}_C_{ij}}$$

where $\text{Ave}_T_{ij}$ is the estimated average transmission time of $t_i$ from $s_j$ to all the neighbors, $e_i$ is instruction size of $t_i$, and $\text{Ave}_C_{ij}$ is the average computing capacity of all the neighboring sites of $s_j$.

If $CCR_{ij}$ is much greater than 1, we say that task $t_i$ is of communication-intensive CCR-type. If $CCR_{ij}$ is much smaller 1, we say that task $t_i$ is of computation-intensive CCR-type. If $CCR_{ij}$ is comparable to 1, we say that task $t_i$ is of neutral CCR-type. The average computing capacity of site $s_j$ is defined as:

$$CC_{j} = \sum_{k=1}^{n_j} cc_{jk} / n_j$$

where $n_j$ is the number of machines within $s_j$, and $cc_{jk}$ is the computing capacity of $m_k$ at $s_j$.

4.2. Task ordering

At each site, various users may submit a number of tasks with different priorities and deadlines. Our ranking strategy takes both task priority and deadline into consideration. The scheduler at each site puts all incoming tasks into a task queue. First, tasks are sorted by decreasing priority, and then by increasing deadline. Sorting by decreasing priority allows scheduling highest priority tasks first can guarantee most of them to be successfully scheduled. Sorting by increasing deadline allows the most urgent tasks to be scheduled first.

4.3. Optimization Model

More precisely, after assigning a task $t_i$ to a machine $m_k$ at site $s_j$, $sl_{pkj}$ which is the length of the slack of the corresponding task slot (denoted as the $p^{th}$ slack) is defined as:

$$sl_{pkj} = st_{pkj} - pb_{pkj}$$

where $st_{pkj}$ is the start time of the $p^{th}$ task slot on $m_k$ at $s_j$, and $pb_{pkj}$ is the pre-bound of the $p^{th}$ task slot on $m_k$ at $s_j$. Next, we define the residual capacity of $m_k$ at $s_j$, denoted as $rc_{kj}$, as the sum of all available slacks:

$$rc_{kj} = \sum_{p=1}^{h} sl_{pkj}$$

where $h$ is the number of task slots residing on $m_k$.

When a task is to be scheduled on a resource site, we postulate that an assignment with the largest sum of squared residual capacity of each machine on this particular site should be preferred because it leaves a larger slack in a machine (or machines) instead of smaller slacks that are scattered over several machines. Larger slacks give added flexibility to accommodate future incoming jobs. To formulate this problem, suppose that $n$ new jobs are to be scheduled on $n$ machines at a resource site. Then, the optimization model for the scheduling problem can be formed as following:

$$X = \sum_{i=1}^{n} x_i^2$$

where $x_i$ denotes the residual capacity of machine $m_i$ and $X$ is the sum of squared residual capacities of each machine. We propose a greedy approach that provably maximizes $X$ at each local step.

4.4. Scheduling

For each site, we rank machines by increasing residual capacity. When a task $t_i$ is to be scheduled on a site, any machine with a smaller residual capacity will have preference to execute $t_i$, if it can meet the deadline. Next, we show that doing so performs the locally “best” assignment for the current task.

Lemma. The machine ranking strategy always improves the objective function at the next step.

1 As an aside, we note that this problem is a type of convex maximization problem.
Proof. Suppose that $t_i$ is to be scheduled on site $s_j$ where $k$ machines $m_1, ..., m_k$ reside and that machines $m_{u}, ..., m_{v}$ are the machines into which $t_i$ can be scheduled to meet the deadline. Since these machines are within the same site, we assume that the execution length of task $t_i$ on every machine is the same, denoted as $c$. Let $z_u, ..., z_v$ be the current residual capacities of these machines. Select machine $m_a$ such that

$$z_a = \min\{z_u, ..., z_v\}. \tag{8}$$

Let $X_a$ be the objective function value Eq. (6) obtained by assigning task $t_i$ to machine $m_a$ at the next step. Let $\beta$ be the contribution to the objective function from all machines other than $m_a$ and $m_a$ in the current assignment. Thus

$$X_n = \beta + (z_n - c)^2 + (z_n)^2,$$

$$X_u = \beta + (z_u)^2 + (z_u - c)^2,$$

giving $X_n - X_u = -2cz_n + 2cz_u + 2c(z_u - z_n) \geq 0.$$

The final inequality follows from Eq. (8). \hfill \Box

To schedule task $t_i$ on a site $s_j$, each machine $m_k$ at $s_j$ will check if $t_i$ can be assigned to meet its deadline. If tasks have already been assigned to $m_k$, task slots of varying length will be available on $m_k$. If no task has been assigned, there are no slots between tasks so that:

$$pb_{pij} = 0 \& \& st_{pij} = 0 \& \& ed_{pij} = 0, \tag{9}$$

where $ed_{pij}$ is the end time of $p^{th}$ task slot on $m_k$ at $s_j$.

First the scheduler checks whether $t_i$ may be inserted after the last task slot, if any. In this way the residual capacity of this machine can be maximized according to Eq. (6). The criterion for $t_i$ to be scheduled and meet its deadline is:

$$e_{ijk} + max(ed_{pij}, c_j) \leq d_i. \tag{10}$$

where $e_{ijk}$ is the estimated execution time of $t_i$ on machine $m_i$ at site $s_j$, $n$ is the last task slot on $m_k$, $c_j$ is the estimated transmission time of $t_i$ from its local site (where $t_i$ is originally submitted) to $s_j$, and $d_i$ is the deadline of $t_i$.

If $t_i$ cannot be scheduled after the last task slot, the scheduler checks whether $t_i$ can be inserted into any slack while meeting its deadline. Slacks are ranked by increasing length. In other words, we always try to assign tasks to smaller slacks first. By doing this, we increase the size of the remaining slacks, which again gives added flexibility to the company in accommodating future incoming jobs. Simulation results also show that inserting tasks into smaller slacks first gives better performance. The criteria to find a feasible slack for $t_i$ are:

$$e_{ijk} + max(pb_{pij}, c_j) \leq st_{pij} \tag{11}$$

If the Eq. (11) is satisfied, we schedule $t_i$ to the $p^{th}$ slack on $m_k$ at $s_j$, and set its start time $s_{ijk}$ to:

$$s_{ijk} = min(d_i, st_{pij}) - e_{ijk}. \tag{12}$$

In general setting task start time to their latest start times leaves large slacks, enabling other tasks to be scheduled within such slacks. Also, if $s_j$ is the local site for $t_i$, the transmission time is ignorable; in other words, $c_j = 0$. The pseudo code of Schedule is shown in Fig. 3.

4.5. Packing

If unscheduled tasks remain after executing Schedule, a packing procedure is executed on each machine of the site. Pack moves all tasks as early as possible. By doing so, Pack creates larger slacks for possible use by unscheduled tasks. Moreover, executing tasks early provides temporal fault-tolerance. The pseudo code of Pack is shown in Fig. 3.

\begin{verbatim}
Schedule
1. for each unscheduled task $t \in Q$
2.   do for each machine $m \in S$
3.     visit slacks in the order of increasing residual capacity
4.     if slack fits within slack // while meeting deadline
5.     Schedule $t$ on $m$ at the latest possible time within the slack
6.     Mark $t$ scheduled
7.     Update $Q$
8. until $t$ is scheduled
end Schedule

Pack
1. for each task $t$ // iterate by decreasing priority
2. Re-Schedule $t$ to the earliest available slack
end Pack
\end{verbatim}

Fig. 3. Schedule and Pack

4.6. Dispatching

For those tasks that cannot be scheduled at their local site, a dispatching phase is needed to forward them to remote sites. We use the same dispatching strategy proposed in [6], a task-resource matched peer-to-peer dispatching that gives better performance and provides scalability. We omit the details in this paper due to space limitation. The pseudo code of Dispatch is shown in Fig. 4.
Dispatch
1. for each unscheduled task \( t \in Q \)
2. for each neighbor \( N \) of \( S \) // visit neighbors in order depending upon CCR-type of \( t \)
3. Send \( t \)'s ticket to \( N \)
4. if \( N \) can successfully schedule \( t \)
5. Send \( t \) to \( N \)
6. Mark \( t \) scheduled
end Dispatch

Fig. 4. Dispatch

4.7. Complexity

Let \( n \) be the number of incoming tasks, \( m \) the number of machines within each site, \( p \) the number of slacks on a single machine, and \( s \) the number of sites. Then, the complexity of Pack is \( O(n) \), and of Dispatch is \( O(n \cdot s) \). The complexity of RCMS’s ranking phase is \( O(n \cdot \log n) \). The complexity of Schedule is \( O(n \cdot p \cdot \log p) \) since the slacks within each machine are to be evaluated in parallel by each machine in a non-blocking fashion. Therefore, the complexity of RCMS is \( O(n \cdot p \cdot \log p) \), assuming \( p \cdot \log p > \log n \) since \( n \) is not exponential in \( p \). We note that the complexity of Sufferage and Min-Min is \( O(msn^2) \). Thus, RCMS is a distributed algorithm with low complexity.

5. Simulations

We conducted extensive simulations based on the San Diego Supercomputer Center (SDSC) SP2 log to evaluate RCMS. The real trace was sampled on a 128-node IBM SP2 (67,665 jobs from April 1998 to April 2000) [15]. To reveal the strengths of RCMS, we compared it with three effective scheduling algorithms, namely, Min-Min, Sufferage, and General Distributed Scheduler (GDS) [6] which considers both task priority and deadline. Moreover, we propose several variations based upon RCMS and compare them with RCMS.

5.1. Simulation parameters

The system parameters in a simulated grid system are chosen to resemble real-world workstations like IBM SP2 nodes. Table 2 summarizes the key parameters of the simulated grid system used in our experiments. We modified a real-world trace by associating each task with a deadline. In this way, we are able to evaluate the impact of deadline constraints on the system performance by adjusting deadline values. Deadlines are synthetically generated in accordance with the task size, such that the grid system is close to its breaking point, where tasks start to miss deadlines. In other words, our simulation results include the performance of all algorithms under the overload condition which is a typical case for real-world applications. Each data point is an average of 30 runs.

The Overall Schedulable Ratio and Hard Schedulable Ratio are used as the performance metrics in the evaluation. These two metrics are defined as:

\[
\text{Overall Schedulable Ratio} = \frac{N_s}{N_{\text{total}}} \tag{13}
\]

where \( N_s \) is the number of scheduled tasks meeting deadlines, and \( N_{\text{total}} \) is the total number of tasks.

\[
\text{Hard Schedulable Ratio} = \frac{N_{s}^{h}}{N_{\text{hard}}} \tag{14}
\]

where \( N_{s}^{h} \) is the number of scheduled hard tasks meeting deadlines, and \( N_{\text{hard}} \) is the total number of hard tasks.

Table 2. Characteristics of system parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value(fixed)-(varied)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of jobs</td>
<td>(6000)-(3000,4000,5000,6000,7000,8000,9000,10000)</td>
</tr>
<tr>
<td>Number of sites</td>
<td>32</td>
</tr>
<tr>
<td>Site processing speed</td>
<td>8*8 nodes</td>
</tr>
</tbody>
</table>

5.2. Variations of RCMS

RCMS is based on the core idea of the Best Fit. There are two other classical algorithms named Worst Fit and First Fit. We proposed two variations of RCMS, namely WRCMS and FRCMS, which correspond to the Worst Fit and First Fit. Among these three algorithms, the ranking, packing, and dispatching phases are the same. The only difference is during the scheduling phase, mainly the machine selection and slack selection strategies. The differences between WRCMS and RCMS are mainly two fold:

1. WRCMS ranks machines at each site by decreasing residual capacity.
2. WRCMS ranks slacks on each machine by decreasing length.

The differences between FRCMS and RCMS are also two fold:

1. FRCMS randomly chooses a machine to schedule a task, without any ranking.
2. Slacks are visited from the first to last, with no ranking step included.

By comparing these three algorithms, we intend to better understand the merits of using the optimization model Eq. (7).
5.3. Performance

The first experiment set was to evaluate the performance of RCMS against WRCMS and FRCMS. For Hard Schedulable Ratio, from Fig. 5, we observe that when the number of tasks is below 5000 all three algorithms can achieve 100% Hard Schedulable Ratio. This is because all of them schedule hard tasks first. Since the total number of tasks is below 5000, the number of hard tasks is smaller than 2000, which is a small enough number for the computing resources to handle. When the number of tasks increases from 6000 to 10000, in other words when the number of hard tasks increases from roughly 2000 to 3400, RCMS performs slightly better (about 1% performance improvement) than WRCMS, and a little better (about 2% performance improvement) than FRCMS. Note that RCMS is not able to schedule all hard tasks when the total number of incoming tasks becomes greater than 5000, which is considered as the overload condition. In normal cases, RCMS is able to provide schedulability guarantee to hard tasks.

![Fig. 5. Hard Schedulable Ratio](image)

**Fig. 5. Hard Schedulable Ratio**

With respect to Overall Schedulable Ratio, as shown in Fig. 6, the performance difference among the three heuristics increases to 8% on average. Since RCMS always schedules tasks first on machines with smaller residual capacity and inserts them into shortest possible slacks, many large slacks could remain to accommodate future firm and soft tasks. The simulation results prove that the optimization model as per Eq. (7) brings significant benefits to RCMS.

![Fig. 6. Overall Schedulable Ratio](image)

**Fig. 6. Overall Schedulable Ratio**

5.4. Performance against Min-Min, Sufferage, and GDS

The second experiment set was to evaluate the performance against Min-Min, Sufferage, and GDS. For hard tasks, from Fig. 7, we observe that RCMS outperforms the other algorithms by over 21%. GDS gives a satisfactory Hard Schedulable Ratio performance since it also schedules hard tasks first. However, RCMS improves the performance under overload conditions by maximizing the slacks in a formalized way. An interesting observation is that when the number of tasks falls below a certain threshold (i.e. 5000) under our simulation configuration, RCMS can achieve 100% Hard Schedulable Ratio. Note that all GDS cannot provide such a performance guarantee, which shows a significant advantage brought by using RCMS.

![Fig. 7. Hard Schedulable Ratio](image)

**Fig. 7. Hard Schedulable Ratio**

Fig. 8 shows that with respect to Overall Schedulable Ratio, RCMS outperforms the other algorithms by about 8% on average. Thus, RCMS not only maximizes the number of hard tasks meeting
deadlines, but it does so with improving the Overall Schedulable Ratio.

Fig. 8. Overall Schedulable Ratio

6. Conclusion

This paper addresses the problem of scheduling mixed independent tasks with different priorities and deadline constraints on computational grids. The scheduling problem is first formulated as a bin packing problem. Next, an algorithm RCMS, which is composed of four phases, is proposed. RCMS applies several innovative strategies and can provide schedulability guarantee to hard tasks while maximizing the total number of incoming tasks to meet deadlines. Moreover, RCMS is a distributed and scalable algorithm that does not need to know the global state of the grid. Simulation results demonstrate efficiency of the proposed approach that RCMS significantly increases both the schedulable ratio of hard tasks and the schedulable ratio of all incoming tasks.

7. References


