Single Restart with Time Stamps for Parallel Task Processing with Known and Unknown Processors

Jaya Prakash Champati, *Member, IEEE,* and Ben Liang, *Fellow, IEEE*

Abstract—We study the problem of scheduling n tasks on $m + m'$ parallel processors, where the processing times on m processors are known while those on the remaining m' processors are not known a priori. This semi-online model is an abstraction of certain heterogeneous computing systems, e.g., with the m known processors representing local CPU cores and the unknown processors representing remote servers with uncertain availability of computing cycles. Our objective is to minimize the makespan of all tasks. We initially focus on the case $m' = 1$ and propose a semi-online algorithm termed Single Restart with Time Stamps (SRTS), which has time complexity $O(n \log n)$. We derive its competitive ratio in comparison with the optimal offline solution. If the unknown processing times are deterministic, the competitive ratio of SRTS is shown to be either always constant or asymptotically constant in practice, respectively in cases where the processing times are independent and dependent on m. A similar result is obtained when the unknown processing times are random. Furthermore, extending the ideas of SRTS, we propose a heuristic algorithm termed SRTS-Multiple (SRTS-M) for the case $m' > 1$. Finally, where tasks arrive dynamically with unknown arrival times, we extend SRTS to Dynamic SRTS (DSRTS) and find its competitive ratio. Besides the proven competitive ratios, simulation results further suggest that SRTS and SRTS-M give superior performance on average over randomly generated task processing times, substantially reducing the makespan over the best known alternatives. Interestingly, the performance gain is more significant for task processing times sampled from heavy-tailed distributions.

Index Terms—Computational offloading, edge computing, mobile cloud computing, opportunistic computing, unknown processing times, task restart, semi-online algorithms

1 Introduction

The problem of parallel task processing on multiple processors has wide-ranging applications in computer science, industrial engineering, and information systems. It is essential to contemporary computing and networking applications, due to the prevalence of multi-core CPUs and the availability of auxiliary resources for computational offloading. Existing studies in parallel task processing can be categorized into three types: *offline*, where the processing times of all tasks on all processors are known a priori; *online*, where no processing time is known until after a task has been processed; and *semi-online*, where some processing time information is known. Scheduling decisions proposed in the research literature generally aim to minimize makespan, i.e., total time to finish the given tasks, or system cost in task processing, or a combination of both. Most of such optimization problems are known to be NP-hard and only approximate solutions are available.

In this work, we study the problem of scheduling computing tasks on $m + m'$ parallel processors, where the task processing times on m processors (*known processors*) are known a priori, and the task processing times on the remaining m' processors (*unknown processors*) are unknown before the tasks are processed. Under this semi-online setting, we are interested in finding a schedule that minimizes the makespan of n tasks that are either all given at time zero or arrive dynamically in time.

The above semi-online scheduling model can be viewed as an abstraction for several important practical systems. The m processors may model parallel CPU cores in a local device (e.g. smartphone, tablet etc.) or processors in a local computing cluster. The unknown processors may represent computational servers whose help is enlisted by the local device or the local cluster. In particular, in mobile cloud computing systems, the unknown processors may be shared virtual machine instances in a public cloud [2], [3]; in Mobile Edge Computing (MEC), they may be MEC servers deployed by a cellular base station [4], [5], [6], [7]; and in cyber foraging/opportunistic computing, they may be neighboring mobile devices or cloudlets onto which the local device offloads its computational tasks [8], [9], [10], [11], [12].

The scenario of not knowing the processing times on the remote processors arises due to various factors. For example, an MEC server is shared between network service tasks and the offloaded user tasks from the subscribing mobile devices. Thus, only a fraction of the MEC server's CPU cycles may be allocated to the mobile user. Similarly, in opportunistic computing, a neighboring mobile device may not dedicate all of its CPU cycles to the offloaded tasks. Furthermore, there can be uncertain delays associated

- *J. P. Champati is affiliated with the Division of Information Science and Engineering, School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, Stockholm, 11428, Sweden. E-mail: jpra@kth.se. B. Liang is affiliated with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, M5S 3G4, Canada. E-mail: liang@ece.utoronto.ca.*
- *This work was supported in part by a grant from the Natural Sciences and Engineering Research Council (NSERC) of Canada. Part of this work has been presented in IEEE INFOCOM 2017 [1].*

with offloading and processing the offloaded tasks. The insights developed from our theoretical model can be used to improve computational offloading in these systems.

Most of the previous studies on parallel processing have focused on either the offline or the fully online scenarios. Offline algorithms are clearly not applicable to our problem. Furthermore, even in the simplest offline setting, makespan minimization is known to be NP-hard [13]. On the other hand, if we ignore the partial knowledge of processing times in our problem, we may use existing online algorithms. In the online setting, if all processors are identical, the well-known *List Scheduling* (LS) algorithm has $\left(2-\frac{1}{m+m'}\right)$ competitive ratio [14]. However, in our case, the unknown processors are not identical to the known ones. In fact, as shown in Section 4.1, List Scheduling has *infinite competitive ratio* for our problem. For online scheduling with non-identical processors, Shmoys *et. al.* has proposed an iterative algorithm that achieves $O(\log n)$ competitive ratio [15]. This algorithm can be applied to our problem. However, as shown in [16], [17], and in Section 7, Shmoys' algorithm does not perform well on average.

We note that List Scheduling and Shmoys' algorithm do not utilize the known processing times in our problem thus leading to their inefficiency. Therefore, our objective is to develop a semi-online algorithm that effectively utilizes both the known and unknown processors, to provide both a provable competitive ratio and satisfactory average performance. Instead of deterministic scheduling such as List Scheduling, we use the approach of *task restarts* similar to [15], where a task scheduled on a processor may be cancelled and re-scheduled possibly on a different processor. Unlike [15], we observe that only one round of restarts is sufficient for our problem. This is similar in design principle to [16], [17], but as explained in Section 2, the problem we consider, the proposed algorithm, and the competitive ratio analysis are substantially different from those in [16], [17].

The main contributions of this work are as follows:

- We first show a negative result, that any semi-online algorithm with a pre-determined scheduling order has infinite competitive ratio. This motivates the need for a more effective dynamic scheduling algorithm.
- We first focus on the important case of $m'=1$, which represents, e.g., the case of mobile cloud computing with m local CPU cores and a more powerful remote cloud server. We propose an efficient Single Restart with Time Stamps (SRTS) algorithm, which has time complexity $O(n\log n)$. We derive its competitive ratio in comparison with the optimal offline solution, for the cases where the known processors are identical and where they have different speeds. If the unknown processing times are deterministic, the competitive ratio of SRTS is shown to be always constant when the processing times are independent of m, and asymptotically constant in practice when the processing times are dependent on m. We obtain a similar result when the unknown processing times are random.
- We extend SRTS to Dynamic SRTS (DSRTS) for the case where tasks arrive dynamically and their arrival times are not known a priori, and show that the resulting algorithm has competitive ratio that differs only by one with the competitive ratio of SRTS.
- Extending the ideas of SRTS, we further propose a heuristic algorithm SRTS-Multiple (SRTS-M), for the case where there are multiple unknown processors, which also has $O(n\log n)$ time complexity.
- To evaluate the average performance of SRTS and SRTS-M, we show using simulation that they provide substantial gains in reducing the makespan over the best known alternatives, for task processing times generated from typical distributions. We further observe that the gains are much more significant for heavy-tailed distributions.

The rest of this paper is organized as follows. In Section 2, we present the related work. The system model is given in Section 3. The SRTS algorithm for identical known processors is presented in Section 4, and its competitive ratio is derived in Section 5. In Section 6, we generalize SRTS to uniform known processors, dynamically arriving tasks, and SRTS-M for the case of multiple unknown processors. We discuss simulation results in Section 7 and conclude in Section 8.

2 Related Work

Scheduling independent tasks on parallel processors is a well-studied problem in theoretical computer science, particularly from the perspective of approximation algorithms. In the following we present relevant works from the literature under offline, online, and semi-online settings.

2.1 Offline and Online Scheduling on Parallel Processors

Even in the simplest *offline* setting, where the processors are *identical*, i.e., for each task the processing times are the same on all processors, the problem is NP-hard [13]. The classical Longest Processing Time (LPT) algorithm forms a list of the tasks in the descending order of their processing times and schedules the next task from the list on whichever processor that becomes idle first. For $m+m'$ identical processors, LPT provides $\left(\frac{4}{3}-\frac{1}{3(m+m')}\right)$-approximation [18]. Other algorithms with various time complexity and approximation ratios are also available in the literature [19], [20]. For the case of non-identical processors, where each task has different processing times, a 2-approximation algorithm was proposed in [21], [22]. Since in our problem the processing times on one processor are not known a priori, none of the above works are applicable.

In the *online* setting, *List Scheduling* (LS) lists the tasks in an arbitrary order and schedules the next task on whichever processor that becomes idle first. It provides a $\left(2-\frac{1}{m+m'}\right)$ competitive ratio for scheduling on $m+m'$ identical processors. LS can be applied to solve our problem, by ignoring the known processing times. However, we will show later that, due to the non-identical processing times among the processors in our problem, a family of deterministic algorithms that include LS has infinite competitive ratio.

Shmoys *et. al.* in [15] considered the general problem of online scheduling of independent tasks on non-identical

processors. Using an approach involving multiple rounds of task restarts, they proposed an $O(\log n)$-competitive online algorithm. Similarly to LS, Shmoys' algorithm can be applied to our problem by simply ignoring the known processing times. However, it has been shown in [16], [17] that the average performance of Shmoys' algorithm can suffer due to the multiple rounds of task restarts. We will show in Section 7 that even a semi-online improvement of this algorithm can give substantially worse average performance than the proposed SRTS algorithm. Furthermore, by judiciously utilizing the known processing times, SRTS achieves a competitive ratio that is independent of n.

2.2 Semi-online Scheduling on Parallel Processors

Studies under semi-online settings are comparatively scarce. Even the definition of semi-online scheduling is not unified. In [23], [24], [25], [26], it refers to the case where only the total processing time of the tasks on each processor is known. Furthermore, all of these works focus on the special case of identical or uniform processors, so they are not applicable to our problem.

To the best of our knowledge, the semi-online setting most similar to ours is in [16], [17], which may be viewed as having one processor with known processing times (which actually was used to model some fixed usage cost in [16], [17]) and m identical processors with unknown processing times. However, the Greedy-One-Restart (GOR) algorithm proposed in [16], [17] cannot be applied to our problem. While GOR schedules tasks on the m *unknown identical* processors using estimated processing times based on the single known processor, SRTS schedules tasks on the m *known* processors directly using the known processing times. Furthermore, the estimation of the unknown processing times for task restarting requires different methods. It depends on m in GOR, while in SRTS the known processing times are directly used as the estimate. Notably, as a result of these differences, the competitive ratio of SRTS is constant under general conditions, in contrast to $O(\sqrt{m})$ for GOR. Furthermore, we consider the case of multiple unknown processors that are non-identical in the SRTS-M algorithm.

2.3 Other Related Works

In mobile cloud computing systems [2], [3], where a mobile device enlists the help of a remote processor in a remote cloud, most current research is focused on the task offloading problem with the objective of minimizing energy, e.g., [27], [28], [29], [30]. In addition, several empirical studies have been conducted on task offloading from a mobile device to remote servers [8], [9], [10], [11], [12]. Furthermore, the hybrid cloud computing architecture, where tasks are offloaded from a local cluster/cloud to a public cloud, has been studied before; see for example [31], [32]. In [33], [34], the authors have studied the problem of scheduling scientific work flows on a subset of available resources in a grid computing system with demand uncertainties. However, all of these works have system models different from ours, and none of them considers makespan as their design objective. In this work, our focus is to provide a general semi-online solution to the problem of makespan minimization in parallel task scheduling, which may be applied to cloud computing and other practical computing and networked systems.

3 System Model

In this section we describe the machine model, task processing times and the makespan minimization problem.

3.1 Processor Model

For clarity of presentation, in Sections 4 and 5 we initially focus on a heterogeneous system comprised of m identical "known" (or "local") parallel processors indexed by $i \in \mathcal{Q} = \{1, \ldots, m\}$ and a single "unknown" (or "remote") processor indexed by $i = m + 1$. Later, in Section 6 we analyse the proposed SRTS algorithm for the case of uniform processors, i.e., the ratio of the processing times of a task on any two processors is a constant. Furthermore, we consider the case of $m' > 1$ unknown processors, for which we propose SRTS-M.

3.2 Processing Times and Task Arrivals

Given n tasks, indexed by $j \in \mathcal{T} = \{1, \ldots, n\}$, our objective is to minimize the makespan to process them. The tasks are assumed independent and non-preemptive. Initially, we focus on the case where all tasks are available at time zero. In Section 6, we will extend this to the case of dynamic task arrivals with unknown arrival times, for which we propose DSRTS.

The processing time of task j on processors in $\mathcal{Q}$ is denoted by a_j and is assumed to be known a priori. This may be obtained, for example, by checking the number of instructions per task and the processor speed. The processing time of task j on processor $m + 1$ is denoted by u_j and is assumed to be unknown until the task has been executed to completion. This may arise in many scenarios of practical interest. For example, a remote server may be shared and only a fraction of the CPU cycles are allocated to the offloaded tasks. There may also be other uncertain delays in offloading and processing the tasks at a remote server. We do not assume any relation between u_j and a_j, but it is important to note that our results can serve as a benchmark to evaluate the performance of algorithms that do consider the relation between u_j and a_j. Since u_j and a_j are independent, the system of local processors and the remote processor falls under the unrelated parallel processors model [22], i.e. the ratio $\frac{a_j}{u_j}$ is not constant and is specific to task j.

Note that even though u_j is unknown, it may be deterministic, i.e., it remains the same independent of when task j is processed. For example, this may model the case where the remote CPU cycles allocated to the tasks do not change frequently. However, u_j may also be random, i.e., it depends on when task j is processed. As shown in Section 5, this distinction is important in performance analysis, since the proposed SRTS algorithm may cancel and then restart a task at a different time. In this work, we consider both cases.

3.3 Semi-Online Scheduling and Competitive Ratio

Let $\mathbf{s}$ denote a schedule and $\mathcal{S}$ denote the set of all possible schedules. The schedule $\mathbf{s}$ decides the placement of a task on one of the known processors $\mathcal{Q}$ and the remote processor $m+1$. Given the set of tasks at time 0, the makespan of a schedule $\mathbf{s}$, denoted by $C_{\max}(\mathbf{s})$, is defined as the time when the processing of the last task is completed. It equals $\max_i\{C_i(\mathbf{s})\}$, where $C_i(\mathbf{s})$ is the completion time of the last task assigned to processor i. We are interested in the following makespan minimization problem $\mathcal{P}$:

$$\underset{\mathbf{s}\in\mathcal{S}}{\text{minimize}} \quad C_{\max}(\mathbf{s}).$$

From Section 2, we see that even for the offline version of $\mathcal{P}$, where all parameter values of the tasks are known at time zero, the problem is NP-hard [13]. We are interested in the more complicated semi-online setting, where u_j are not known a priori.

The efficacy of an online algorithm is often measured by its competitive ratio in comparison with an optimal offline algorithm. We use the same measure for semi-online algorithms as well. Let $\{P,\{u_j\}\}$ be a problem instance of $\mathcal{P}$, where $P=\{m,n,\{a_j\}\}$. Let $\mathbf{s}(P,\{u_j\})$ be the schedule given by a semi-online algorithm and $\mathbf{s}^*(P,\{u_j\})$ be the schedule given by an optimal offline algorithm. If u_j are deterministic, then the problem instance $\{P,\{u_j\}\}$ is a set of constants and an optimal offline algorithm outputs the minimum makespan $C_{\max}(\mathbf{s}^*(P,\{u_j\}))$. In this case the semi-online algorithm is said to have a competitive ratio θ if

$$\sup_{\forall\{P,\{u_j\}\}} \frac{C_{\max}(\mathbf{s}(P,\{u_j\}))}{C_{\max}(\mathbf{s}^*(P,\{u_j\}))} \leq \theta. \tag{1}$$

If u_j are random, it is not straightforward to extend the competitive ratio definition in (1) for the proposed SRTS algorithm, as it restarts some tasks. Toward this end, in the following we define a more general competitive ratio measure for this case. Let $\{P,F_u\}$ be a problem instance of $\mathcal{P}$, where F_u is the joint distribution of $\{u_j\}$. For a given P, let $\{u_j^{(1)}\}$ denote the task processing times without restart. Note that the joint distribution of $\{u_j^{(1)}\}$ will be F_u. Let $\{u_j^{(2)}\}$ denote the task processing times when tasks are restarted. We consider $\{u_j^{(2)}\}$ to potentially have a different distribution from F_u and are possibly correlated to $\{u_j^{(1)}\}$. Let $\mathbf{s}^{\text{SRTS}}$ denote the schedule under SRTS, then the expected makespan under SRTS is given by $\mathbb{E}[C_{\max}(s^{\text{SRTS}}(P,\{u_j^{(1)}\},\{u_j^{(2)}\}))]$, where the expectation is over the joint distribution of $\{u_j^{(1)}\}$ and $\{u_j^{(2)}\}$. For random u_j, SRTS is said to have a competitive ratio θ if

$$\sup_{\forall\{P,F_u\}} \frac{\mathbb{E}[C_{\max}(s^{\text{SRTS}}(P,\{u_j^{(1)}\},\{u_j^{(2)}\})]}{\mathbb{E}[C_{\max}(s^*(P,\{u_j\}))]} \leq \theta. \tag{2}$$

4 Single Restart with Time Stamps (SRTS)

In this section we focus on the important case of $m'=1$. We first present our design consideration for SRTS, then describe the algorithm details, and finally present an illustrative example to explain the working of SRTS.

4.1 Design Considerations

In this subsection we explain the failure of some existing algorithms for our problem and derive insights into the design of SRTS.

4.1.1 *Failure of algorithms with pre-determined scheduling order*

We note that the celebrated LS algorithm can be used to solve $\mathcal{P}$ as it does not require the processing times of the tasks on any processor. Also, one can extend the LPT algorithm to solve $\mathcal{P}$ by sorting tasks based on the known a_j values. In the rest of this paper we term this algorithm Semi-Online LPT (SO-LPT). Both LS and SO-LPT belong to the family of algorithms with a *pre-determined scheduling order*, which is formally defined as algorithms that rank the tasks according to some rule and then schedule them one after another in that fixed order. In the following, we study the performance of these algorithm.

In Section 2, we have noted that when all processors are identical, LS has a constant competitive ratio. Also, if all processors are identical and the processing times of the tasks are known, then LPT has $\frac{4}{3}$ approximation ratio [18]. Since our problem model has m known identical processors with only one unknown processor, one may expect that there exists some deterministic scheduling algorithm that gives a low competitive ratio. However, in the following theorem, we observe that the family of all algorithms with a per-determined scheduling order are highly ineffective in the worst case. Therefore, we need a more flexible dynamic scheduling approach in our design of SRTS.

Theorem 1. *Any algorithm with pre-determined scheduling order has infinite competitive ratio with respect to $\mathcal{P}$.*

Proof. To prove the result it is sufficient to identify a problem instance where the algorithm gives a makespan whose ratio over the optimal makespan is infinite. Consider the following family of problem instances: $u_1=\alpha$, where $\alpha>1$ is an arbitrary constant, $u_j=1,\forall j\in\{2,\ldots,n\}$, and $a_j=1,\forall j$. We further assume n is a multiple of $m+1$. Since u_j are unknown, any algorithm using some pre-determined order to schedule the tasks can only use the knowledge of a_j. However, since all a_j are equal, the tasks cannot be differentiated by such an algorithm. Therefore, in the worst case, it may schedule task 1 on processor $m+1$. This will result in a makespan of at least α.

Note that in the above family of problem instances, the processing time of any task on any processor is at least 1. Since there are n tasks and $m+1$ processors, the offline optimal makespan cannot be less than $\frac{n}{m+1}$. Furthermore, this lower bound can be achieved by the following schedule. Use LS to schedule tasks from $\{1,\ldots,n-\frac{n}{m+1}\}$ on processors $i\in\mathcal{Q}$ and schedule all other tasks, $\{n-\frac{n}{m+1}+1,\ldots,n\}$, on processor $m+1$. Therefore, the offline optimal makespan is $\frac{n}{m+1}$ and the makespan ratio of any algorithm with pre-determined scheduling order is at least $\alpha(m+1)/n$. The result follows as the value of α can be chosen to be arbitrarily large. □

4.1.2 *Inefficiency of multiple rounds of restarts*

In Section 2, we have noted that Shmoys' online algorithm is the only existing algorithm that can be applied

to solve $\mathcal{P}$ with a provable competitive ratio. Shmoys' algorithm initially estimates the unknown processing times of the tasks and then uses any ρ-approximation offline algorithm to schedule them. Tasks that are not completed within the estimated time are cancelled and rescheduled using an increased estimate for the unknown processing times and the same offline algorithm. The procedure is repeated until all tasks are completed. This algorithm has $(4\rho \log n + 4\rho \log 2\rho + 1)$ competitive ratio [15]. However, its average performance may be unsatisfactory [16], [17], and its competitive ratio still depends on n.

One might consider improving Shmoys' algorithm to a semi-online version to solve $\mathcal{P}$, by incorporating the information about known processing times a_j. We term this improved version *Semi-Online Shmoys* (SO-Shmoys) in the rest of this paper. In SO-Shmoys, we use a_j as the initial estimate of the unknown processing time u_j, and LPT as the offline component algorithm. For each iteration, the estimated processing time is doubled. In iteration k, since LPT is applied to an offline problem where the processing time of a task is a_j on the first m processors and $2^k a_j$ on processor $(m+1)$, it yields 2 approximation [35] for all k. Thus, overall SO-Shmoys remains $O(\log n)$-competitive, and its average performance is improved. However, as shown in Section 7, we observe that SO-Shmoys does not perform better than SO-LPT in terms of average performance. This is due to the multiple rounds of task restarts, each penalizing the makespan, since the time already spent on processing a cancelled task is wasted.

Therefore, in SRTS we use only a single round of task restarts. Cancelling a task with large u_j on processor $m+1$ may allow some tasks that have smaller u_j values to be scheduled on that processor. At the same time, we avoid the wastage of time in cancelling a task more than once. As shown in Sections 5 and 7, our new design achieves both a small competitive ratio and superior average performance. A detailed description of SRTS is given below.

4.2 SRTS Algorithm Description

SRTS is comprised of two iterations. In the initial iteration, it first uses a_j as the estimate for the processing time of task j on the unknown processor $m+1$. It forms a list according to the ascending order of a_j. When processor $m+1$ becomes idle, it schedules the next available task from the *end* of the list. If the task is not completed within duration a_j, it cancels the task and sets it aside. Whenever a processor in the known processor set $\mathcal{Q}$ becomes idle, it schedules the next available task from the *start* of the list. We note that the above scheduling order of tasks is advantageous for tasks that incur large a_j and small u_j values.

After going through all tasks in the first iteration above, those tasks that are cancelled are again sorted, and a list is formed in the ascending order of a_j. In the second iteration, this list is scheduled using the same procedure as above, but this time we do not cancel a task, unless it is simultaneously scheduled on two processors. Note that in both iterations some tasks may be scheduled on both processor $m+1$ and some processor in $\mathcal{Q}$. In such a case we cancel the task on one processor if it is either *completed or cancelled* on another processor first.

SRTS can be readily implemented in practice by a scheduler in a local device or a local cluster. For example, this can be achieved by assigning time stamps to the tasks that are offloaded. A remote processor looks at the time stamp of a task to determine when to discard it. The local scheduler decides to restart an offloaded task if it does not receive an acknowledgement or the output of the task within the duration specified by time stamp.

The details of the algorithm are presented in Algorithm 1, where $l = 1$ or 2 indicates the iteration number. We note that SRTS runs in $O(n \log n)$ time due to the need for sorting n tasks. We use $\mathbf{s}^{\text{SRTS}}$ to denote the resultant schedule.

Algorithm 1: Single Restart with Time Stamps (SRTS)

$\mathcal{T}^{(1)} = \mathcal{T}$
for $l = 1$ to 2 **do**
  Sort $\mathcal{T}^{(l)}$ in the ascending order of a_j. WLOG, re-index tasks such that $a_1 \leq a_2 \leq \ldots \leq a_{|\mathcal{T}^{(l)}|}$.
  $j_1 = |\mathcal{T}^{(l)}|$, $j_0 = 0$
  Start processing task j_1 on processor $m+1$
  if $l = 1$ **then**
    Cancel task j_1 if its execution time exceeds a_{j_1} and include it in $\mathcal{T}^{(l+1)}$
  end if
  for $k = 1$ to $\min\{m, |\mathcal{T}^{(l)}|\}$ **do**
    $j_0 = j_0 + 1$
    Start processing task j_0 on processor k.
  end for
  while $\mathcal{T}^{(l)} \neq \emptyset$ **do**
    Wait until a processor becomes idle
    if the idle processor is $\hat{i} \in \mathcal{Q}$ **then**
      Let task j be the last task completed on $\hat{i}$
      Cancel task j if it is scheduled on processor $m+1$
      $\mathcal{T}^{(l)} = \mathcal{T}^{(l)} \backslash \{j\}$
      $j_0 = j_0 + 1$
      If task j_0 is not completed or cancelled yet, schedule it on processor $\hat{i}$
    else if the idle processor is $m+1$ **then**
      Cancel task j_1 if it is scheduled on some processor from $\mathcal{Q}$
      $\mathcal{T}^{(l)} = \mathcal{T}^{(l)} \backslash \{j_1\}$
      $j_1 = j_1 - 1$
      If task j_1 is not completed yet, schedule it on processor $m+1$
      if $l = 1$ **then**
        Cancel task j_1 if its execution time exceeds a_{j_1} and include it in $\mathcal{T}^{(l+1)}$
      end if
    end if
  end while
end for

4.3 Illustrative Examples

Example 1: In the following, we explain the working of SRTS using the following family of problem instances considered in the proof of Theorem 1: $u_1 = \alpha > 1$, $u_j = 1, \forall j \in$

$\{2, \ldots, n\}$, and $a_j = 1, \forall j$. For simplicity of illustration, we further assume that n is a multiple of $m+1$. Since $a_j = 1, \forall j$, SRTS do not differentiate the tasks. Let us consider the worst case scenario, where task 1 is scheduled on processor $m+1$ in the first iteration of SRTS. Note that in the first iteration of SRTS, any task scheduled on processor $m+1$ is processed for a duration of $\min\{a_j, u_j\}$, which is equal to 1 for all tasks. Therefore, task 1 will be cancelled after a duration of 1. At the end of the first iteration, $n - \frac{n}{m+1}$ tasks will be completed on the processors in $\mathcal{Q}$ and $\frac{n}{m+1} - 1$ of them will be completed on processor $m+1$, with task 1 being cancelled. Then, in the second iteration of SRTS, task 1 will be completed on some processor in $\mathcal{Q}$.

In the first iteration, the $n - \frac{n}{m+1}$ tasks on the processors in $\mathcal{Q}$ require a duration of $\frac{1}{m}\left(n - \frac{n}{m+1}\right) = \frac{n}{m+1}$ on each processor. On processor $m+1$, the duration is also $\frac{n}{m+1}$. The duration of the second iteration is 1. Therefore, the makespan of SRTS for these problem instances is $\frac{n}{m+1} + 1$, which is independent of the unknown processing time α. Note that, since the processing time of any task on any processor is at least 1, the offline optimal makespan cannot be less than $n/(m+1)$. This example illustrates that, by restarting a task that has larger u_j value, SRTS can effectively limit the impact of that task on the makespan.

Example 2: We consider another example where there is only one local processor and the remote processor. Three tasks, indexed $\{1, 2, 3\}$, arrive at the local processor with local processing times $\{5, 7, 8\}$ and the unknown remote processing times $\{3, 1, 20\}$, respectively. For this set of tasks the schedule under SRTS is shown in Figure 4.3. The task indices are labelled in red and are italicised, and the processing durations are labelled in black. Since the known processing times of the tasks are already sorted, in the first iteration, SRTS schedules task 1 on the local processor and task 3 on the remote processor with estimated processing time of 8 units. When Task 1 is finished at time 5, task 2 is scheduled on the local processor. Since task 3 is not finished within the estimated time of 8 units, it will be cancelled on the remote processor. Now that the remote processor is idle and task 2 is not finished, SRTS schedules it on the remote processor and thus task 2 is simultaneously scheduled on both processors. The first iteration ends when the remote processor finishes execution of task 2, at which point task 2 will be cancelled on the local processor. In the second iteration the cancelled task 3 is executed to completion on the local processor. Thus, SRTS achieves a makespan of 17 units for this example. We note that LS results in a makespan of 20 units, in the worst case, by scheduling task 3 on the remote processor and the other two tasks on the local processor.

5 Competitive Ratio Analysis

In this section, we first consider the case of deterministic u_j and derive a competitive ratio θ_1 for SRTS. Then, we extend the result to the case of random u_j, showing that the same competitive ratio θ_1 holds with only minor modification.

5.1 Deterministic u_j

In each iteration l of Algorithm 1, where $l = 1$ or 2, we consider the following intermediate outcome of SRTS that will be used extensively in our analysis and proofs. Let $\mathbf{s}_l$ denote the intermediate schedule in iteration l obtained by breaking the loop in Line 13 of Algorithm 1 as soon as j_0 becomes equal to j_1. We note that $\mathbf{s}_l$ is a schedule over the set $\mathcal{T}^{(l)}$, and all the tasks from $\mathcal{T}^{(l)}$ will be scheduled at least once under $\mathbf{s}_l$. To understand this, in the while loop from Line 13 of Algorithm 1, when $j_0 = j_1 - 1$, all the $|\mathcal{T}^{(l)}|$ tasks should have been scheduled on some processor. Now, any more iterations in the while loop will only result in scheduling a task that is already scheduled on processor $m+1$ onto some processor in $\mathcal{Q}$, or vice-versa. Since under $\mathbf{s}_l$ the while loop breaks when $j_0 = j_1$, there will be only one task that is scheduled on both processor $m+1$ and some processor in $\mathcal{Q}$. This will be the last task scheduled by $\mathbf{s}_l$ in iteration l, and we denote it by $q^{(l)} = j_0 = j_1$.

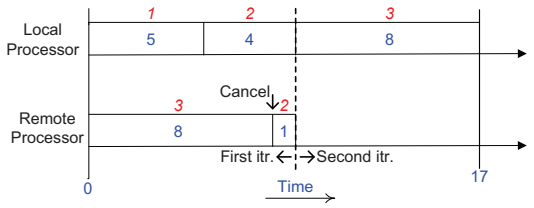

Fig. 1. SRTS schedule for an example problem.

We refer to the time to process the set of tasks $\mathcal{T}^{(l)}$ in iteration l as the *schedule length* of this iteration, denoted by $C^{(l)}_{\max}$. In the rest of this paper, to differentiate the terms with respect to $\mathbf{s}^{\text{SRTS}}$ and $\mathbf{s}_l$, we append onto them the labels of $(\mathbf{s}^{\text{SRTS}})$ and $(\mathbf{s}_l)$, respectively. We note that in iteration l, the schedule produced by $\mathbf{s}^{\text{SRTS}}$ improves on $\mathbf{s}_l$. To see this, observe that $\mathbf{s}_l$ stops scheduling when $j_0 = j_1$. The step $j_0 = j_1$ also occurs under $\mathbf{s}^{\text{SRTS}}$ in both iterations. However, $\mathbf{s}^{\text{SRTS}}$ may not stop at this step. If processor $m+1$ is faster and completes task $q^{(l)} - k$ first, where $k \in \{0, 1, \ldots, \min\{q^{(l)}, m\} - 1\}$, then $\mathbf{s}^{\text{SRTS}}$ schedules task $q^{(l)} - k - 1$, if it is not completed yet, onto processor $m+1$. This will result in a schedule length no longer than that given by $\mathbf{s}_l$, i.e., $C^{(l)}_{\max}(\mathbf{s}^{\text{SRTS}}) \leq C^{(l)}_{\max}(\mathbf{s}_l)$.

Note that, in the analysis and proofs that follow, we do not explicitly mention problem instance $\{P, \{u_j\}\}$, as the results are valid over all possible problem instances. Also, we simply use $\mathbf{s}^*$ to denote an optimal offline schedule and $C^*_{\max}$ to denote the offline optimal makespan.

In Lemma 1, using load balancing arguments we establish a relation between $C^{(l)}_{\max}(\mathbf{s}_l)$ and the known processing times a_j. Recall that under schedule $\mathbf{s}_l$, $q^{(l)}$ is the last task scheduled on processor $m+1$ and some processor, say $\hat{i}$, from $\mathcal{Q}$. Let $C^{(l)}_{\max}(\mathbf{s}_l) = C^{(l)}_{\bar{i}}(\mathbf{s}_l)$ for some processor $\bar{i} \in \mathcal{Q} \cup \{m+1\}$.

Lemma 1.

$$mC^{(l)}_{max}(\mathbf{s}_l) \leq \sum_{j \in \mathcal{T}^{(l)}} a_j + (m-1)a_{q^{(l)}},$$

where task $q^{(l)}$ is the last task scheduled, in iteration l, under schedule $\mathbf{s}_l$.

Proof. In iteration l, let $C^{(l)}_i$ denote the schedule length, and $\mathcal{T}^{(l)}_i$ denote the set of tasks scheduled on processor i.

We note that when a task is scheduled on two processors simultaneously, it will be included in the task set of the processor on which it is completed or cancelled first. We consider the following cases.

Case 1: $\bar{i} = m+1$. For this case, task $q^{(l)}$ is scheduled both on processor $m+1$ and processor $\hat{i}$, but completed or cancelled first on processor $m+1$. Therefore, $C^{(l)}_{\max}(\mathbf{s}_l)$ should be smaller than the sum of the processing times of tasks scheduled on processor $\hat{i}$ plus $a_{q^{(l)}}$, i.e.,

$$C^{(l)}_{\max}(\mathbf{s}_l) \leq \sum_{j \in \mathcal{T}^{(l)}_{\hat{i}}(\mathbf{s}_l)} a_j + a_{q^{(l)}}. \tag{3}$$

Also, at time $\sum_{j \in \mathcal{T}^{(l)}_{\hat{i}}(\mathbf{s}_l)} a_j$, all the processors in $\mathcal{Q}\backslash\{\hat{i}\}$ should be busy executing some task, since otherwise the task $q^{(l)}$ would have been scheduled on that processor which is idle before this time. Therefore,

$$\begin{aligned} &\sum_{j \in \mathcal{T}^{(l)}_{\hat{i}}(\mathbf{s}_l)} a_j \leq \sum_{j \in \mathcal{T}^{(l)}_{i}(\mathbf{s}_l)} a_j, \forall i \in \mathcal{Q}\backslash\{\hat{i}\} \\ &\Rightarrow C^{(l)}_{\max}(\mathbf{s}_l) \leq \sum_{j \in \mathcal{T}^{(l)}_{i}(\mathbf{s}_l)} a_j + a_{q^{(l)}}, \forall i \in \mathcal{Q}\backslash\{\hat{i}\}. \end{aligned} \tag{4}$$

In the second inequality above, we have used (3). Since task $q^{(l)}$ is completed or cancelled first on processor $m+1$, $q^{(l)} \in \mathcal{T}^{(l)}_{m+1}(\mathbf{s}_l)$. Note that in Algorithm 1, the tasks are listed in the ascending order of a_j and then the tasks from the start of the list are scheduled on processors in $\mathcal{Q}$. This implies $\cup_{i \in \mathcal{Q}} \mathcal{T}^{(l)}_i(\mathbf{s}_l) = \{1, \ldots, q^{(l)}-1\} \subseteq \mathcal{T}^{(l)}$. Now, summing (3) and (4) for all i in $\mathcal{Q}\backslash\{\hat{i}\}$, we obtain

$$\begin{aligned} mC^{(l)}_{\max}(\mathbf{s}_l) &\leq \sum_{i \in \mathcal{Q}} \sum_{j \in \mathcal{T}^{(l)}_i(\mathbf{s}_l)} a_j + ma_{q^{(l)}} \\ &= \sum_{j \in \cup_{i \in \mathcal{Q}} \mathcal{T}^{(l)}_i(\mathbf{s}_l)} a_j + ma_{q^{(l)}} \\ &= \sum_{j=1}^{q^{(l)}-1} a_j + ma_{q^{(l)}} \\ &\leq \sum_{j \in \mathcal{T}^{(l)}} a_j + (m-1)a_{q^{(l)}}. \end{aligned} \tag{5}$$

Case 2: $\bar{i} = \hat{i}$. Since $q^{(l)}$ is the last task scheduled on processor $\hat{i}$, at time $C^{(l)}_{\max} - a_{q^{(l)}}$ all processors in $\mathcal{Q}\backslash\{\hat{i}\}$ should be busy executing some task. Therefore,

$$C^{(l)}_{\max}(\mathbf{s}_l) \leq \sum_{j \in \mathcal{T}^{(l)}_i} a_j + a_{q^{(l)}}, \forall i \in \mathcal{Q}\backslash\{\hat{i}\}. \tag{6}$$

Summing (6) and $C^{(l)}_{\max}(\mathbf{s}_l) = \sum_{j \in \mathcal{T}^{(l)}_{\hat{i}}} a_j$, and noting that $\cup_{i \in \mathcal{Q}} \mathcal{T}^{(l)}_i = \{1, \ldots, q^{(l)}\} \subseteq \mathcal{T}^{(l)}$ for this case, we obtain the intended result.

Case 3: $\bar{i} \notin \{\hat{i}, m+1\}$. We claim that for this case task $q^{(l)}$ is completed or cancelled first on processor $m+1$. Note that processors from $\mathcal{Q}$ are identical, and tasks are sorted in the ascending order of a_j and re-indexed such that $a_1 \leq a_2 \leq \ldots \leq a_{q^{(l)}}$. This implies that task $q^{(l)}$ has the largest processing time among the tasks scheduled on processors in $\mathcal{Q}$, and it has the latest starting time. If task $q^{(l)}$ were completed on processor $\hat{i}$, then $C^{(l)}_{\max}(\mathbf{s}_l) = C^{(l)}_{\hat{i}}(\mathbf{s}_l)$, which would be a contradiction for this case since $\bar{i} \neq \hat{i}$.

Now, completing task $q^{(l)}$ on processor $\hat{i}$ would have increased the schedule length. Further, task $q^{(l)}$ is scheduled on processor $\hat{i}$ because at the time when processor $\hat{i}$ becomes idle and $q^{(l)}$ is the next task to be scheduled, all other processors are busy executing some task. The above two observations imply that scheduling and completing task $q^{(l)}$ on any of the processors in $\mathcal{Q}$ would have increased the schedule length. This results in same inequalities as in Case 1, and using the same manipulation we can obtain the intended result. □

In the first iteration of SRTS, task j scheduled on processor $m+1$ is processed for duration $\min\{u_j, a_j\}$, since it is cancelled if its processing duration exceeds a_j. We use this fact and Lemma 1 to derive an upper bound for $C^{(1)}_{\max}(\mathbf{s}^{\text{SRTS}})$, which is given in Lemma 2.

Lemma 2.

$$C^{(1)}_{max}(\mathbf{s}^{SRTS}) \leq \min\left\{2 + \frac{\beta_{max}-2}{m+1}, m+1\right\} C^*_{max},$$

where $\beta_{max} = \max_j \frac{a_j}{u_j}$.

Proof. We use $C^{(1)}_i$ and $\mathcal{T}^{(1)}_i$ as defined in the proof of Lemma 1.

Consider $m+1$ hypothetical processors on which the processing time of any task j is $\min\{a_j, u_j\}$. Assume that each task j can be arbitrarily divided into smaller chunks and processed on these processors. Then the minimum makespan in this case is $\frac{1}{m+1}\sum_{j \in \mathcal{T}} \min\{a_j, u_j\}$. This implies $C^*_{\max} \geq \frac{1}{m+1}\sum_{j \in \mathcal{T}} \min\{a_j, u_j\}$.

In the first iteration of SRTS, a task j scheduled on processor $m+1$ is executed for the duration $\min\{a_j, u_j\}$. SRTS does better than a schedule that statically schedules all tasks on processor $m+1$, because it greedily schedules the tasks using all processors and simultaneously execute some tasks between processor $m+1$ and other processors for a possible reduction in makespan. This implies $C^{(1)}_{\max}(\mathbf{s}^{SRTS}) \leq \sum_{j \in \mathcal{T}} \min\{a_j, u_j\}$. Therefore, $C^{(1)}_{\max}(\mathbf{s}^{SRTS}) \leq (m+1)C^*_{\max}$.

To obtain the other part of the bound, we first establish a new lower bound for $C^*_{\max}$. Let $\mathcal{T}^*_i \subseteq \mathcal{T}$ denote the set of tasks scheduled on processor i under an optimal schedule $\mathbf{s}^*$. We have the following inequalities:

$$C^*_{\max} \geq \sum_{j \in \mathcal{T}^*_i} a_j, \forall i \in \mathcal{Q}, \tag{7}$$

and

$$\begin{aligned} C^*_{\max} &\geq \sum_{j \in \mathcal{T}^*_{m+1}} u_j \geq \min_j \left(\frac{u_j}{a_j}\right) \sum_{j \in \mathcal{T}^*_{m+1}} a_j \\ &\Rightarrow \beta_{\max} C^*_{\max} \geq \sum_{j \in \mathcal{T}^*_{m+1}} a_j. \end{aligned} \tag{8}$$

Summing the inequalities in (7), for all $i \in \mathcal{Q}$, and (8), we get

$$(m + \beta_{\max})\, C^*_{\max} \geq \sum_{j=1}^{n} a_j. \tag{9}$$

Since $C^{(1)}_{\max}(\mathbf{s}^{SRTS}) \leq C^{(1)}_{\max}(\mathbf{s}_1)$, it is sufficient to establish the other part of the bound for $C^{(1)}_{\max}(\mathbf{s}_1)$. Now, using

$l = 1$ in Lemma 1, we have $mC^{(1)}_{\max}(\mathbf{s}_1) \leq \sum_{j\in\mathcal{T}^{(1)}} a_j + (m-1)a_{q^{(1)}}$. In the following we further improve this inequality. We claim that

$$(m+1)C^{(1)}_{\max}(\mathbf{s}_1) \leq \sum_{\mathcal{T}^{(1)}} a_j + ma_{q^{(1)}}. \tag{10}$$

To prove the claim we consider the following cases. Recall that $\bar{i}$ is the processor that finishes last and $\hat{i}$ is the processor from $\mathcal{Q}$ on which $q^{(1)}$ is scheduled.

Case 1: $\bar{i} \in \{\hat{i}, m+1\}$. For this case we claim that

$$C^{(1)}_{\max}(\mathbf{s}_1) \leq \sum_{j\in\mathcal{T}^{(1)}_{m+1}(\mathbf{s}_1)} a_j. \tag{11}$$

Note that in the first iteration, any task j is run for at most a_j duration. This is because SRTS cancels the execution of task j if its processing time on processor $m+1$ exceeds a_j. Clearly, (11) holds for $\bar{i} = m+1$.

If $\bar{i} = \hat{i}$, then task $q^{(1)}$ is finished on $\hat{i}$ and is cancelled on processor $m+1$. In other words, if task $q^{(1)}$ is executed on processor $m+1$ until it is finished or it exceeds time duration a_j, then $C^{(1)}_{m+1}(\mathbf{s}_1)$ would have exceeded $C^{(1)}_{\max}(\mathbf{s}_1)$. Since in this case $C^{(1)}_{m+1}(\mathbf{s}_1)$ is at most $\sum_{j\in\mathcal{T}^{(1)}_{m+1}(\mathbf{s}_1)} a_j$, we conclude that (11) holds for $\bar{i} = \hat{i}$. Now, summing (11) and (5) for $l = 1$, and noting that $\mathcal{T}^{(1)}_{m+1}(\mathbf{s}_1) \subseteq \{q^{(1)}, q^{(1)}+1, \ldots, n\}$, we obtain (10).

Case 2: $\bar{i} \in \mathcal{Q}\backslash\{\hat{i}\}$. We note that in this case $q^{(1)} - 1$ is the last task that finishes on $\bar{i}$, because under $\mathbf{s}_1$ this is the latest task scheduled on a processor from $\mathcal{Q}$ before $q^{(1)}$. At time $C^{(1)}_{\max}(\mathbf{s}_1) - a_{q^{(1)}-1}$, all the processors $\{\mathcal{Q}\backslash\{\bar{i}\}\} \cup \{m+1\}$ would have been busy. Therefore, we have

$$C^{(1)}_{\max}(\mathbf{s}_1) = \sum_{j\in\mathcal{T}^{(1)}_{\bar{i}}(\mathbf{s}_1)} a_j,$$

$$C^{(1)}_{\max}(\mathbf{s}_1) - a_{q^{(1)}-1} \leq \sum_{j\in\mathcal{T}^{(1)}_{i}(\mathbf{s}_1)} a_j,\ \forall i \in \{\mathcal{Q}\backslash\{\bar{i}\}\} \cup \{m+1\}.$$

In the last inequality above we have used the fact that the processing time of a task j scheduled on processor $m+1$ does not exceed a_j. Summing the above two inequalities for all $i \in \mathcal{Q} \cup \{m+1\}$, noting that $a_{q^{(1)}-1} \leq a_{q^{(1)}}$, and $\cup_i \mathcal{T}^{(1)}_i(\mathbf{s}_1) = \mathcal{T}^{(1)}$ we obtain (10).

Now, using (9) in (10) we obtain

$$C^{(1)}_{\max}(\mathbf{s}_1) \leq \left(\frac{m}{m+1} + \frac{\beta_{\max}}{m+1}\right) C^*_{\max} + \frac{m}{m+1} a_{q^{(1)}}.$$

To complete the proof, it is sufficient to compare $C^*_{\max}$ and $a_{q^{(1)}}$. Toward this end, we first note from the above inequality that, if $C^*_{\max} \geq a_{q^{(1)}}$, then the lemma is true. Next, we argue that if $C^*_{\max} < a_{q^{(1)}}$, then $C^*_{\max} \geq C^{(1)}_{\max}(\mathbf{s}_1)$, in which case the lemma is already true. Recall that we re-indexed the task such that $a_1 \leq a_2 \leq \ldots \leq a_n$. Suppose $C^*_{\max} < a_{q^{(1)}}$, then the only possibility is that, under the optimal schedule $\mathbf{s}^*$, we have $\mathcal{T}^*_{m+1} \supseteq \mathcal{T}' = \{q^{(1)}, q^{(1)}+1, \ldots, n\}$, i.e., all the tasks from $\mathcal{T}'$ should have been executed on processor $m+1$. Otherwise, if one of those tasks were scheduled by $\mathbf{s}^*$ on a processor in $\mathcal{Q}$, then we would have $C^*_{\max} \geq a_{q^{(1)}}$. Further, we note that $\mathcal{T}^{(1)}_{m+1}(\mathbf{s}_1) \subseteq \mathcal{T}'$, as $\mathbf{s}_1$ schedules tasks from the end of the list $\{1, 2, \ldots, n\}$ on processor $m+1$ and task $q^{(1)}$ is the last task that is completed or cancelled on processor $m+1$. From the above analysis we have

$$\begin{aligned} C^{(1)}_{\max}(\mathbf{s}_1) &\leq \sum_{j\in\mathcal{T}^{(1)}_{m+1}(\mathbf{s}_1)\cup\{q^{(1)}\}} \min\{a_j, u_j\} \leq \sum_{j\in\mathcal{T}'} \min\{a_j, u_j\} \\ &\leq \sum_{j\in\mathcal{T}^*_{m+1}} u_j \leq C^*_{\max}. \end{aligned}$$

In the first inequality, we have again used the fact that in the first iteration of SRTS, a task j is executed for duration $\min\{a_j, u_j\}$ on processor $m+1$. Hence the result. □

A task j scheduled in the second iteration has the property $u_j > a_j$. Using this fact along with Lemma 1, we arrive at Lemma 3.

Lemma 3. *$C^{(2)}_{max}(\mathbf{s}^{SRTS}) \leq 2C^*_{max}$*

Proof. Since the tasks scheduled from $\mathcal{T}^{(2)}$ are cancelled in the first iteration, we have $u_j > a_j$, for all $j \in \mathcal{T}^{(2)}$. Therefore,

$$C^*_{\max} \geq \frac{1}{m+1} \sum_{j\in\mathcal{T}^{(2)}} \min\{a_j, u_j\} = \frac{1}{m+1} \sum_{j\in\mathcal{T}^{(2)}} a_j. \tag{12}$$

Using $l = 2$ in Lemma 1 and noting that $C^{(2)}_{\max}(\mathbf{s}^{\text{SRTS}}) \leq C^{(2)}_{\max}(\mathbf{s}_2)$, we have

$$mC^{(2)}_{\max}(\mathbf{s}^{\text{SRTS}}) \leq \sum_{j\in\mathcal{T}^{(2)}} a_j + (m-1)a_{q^{(2)}}. \tag{13}$$

Using (12) and $C^*_{\max} \geq a_j$, for all $j \in \mathcal{T}^{(2)}$, in (13), we obtain

$$\begin{aligned} &mC^{(2)}_{\max}(\mathbf{s}^{\text{SRTS}}) \leq (m+1)C^*_{\max} + (m-1)C^*_{\max} \\ &\Rightarrow C^{(2)}_{\max}(\mathbf{s}^{\text{SRTS}}) \leq 2C^*_{\max}. \end{aligned}$$

□

Noting that $C_{\max}(\mathbf{s}^{\text{SRTS}}) = C^{(1)}_{\max}(\mathbf{s}^{\text{SRTS}}) + C^{(2)}_{\max}(\mathbf{s}^{\text{SRTS}})$, the following theorem immediately follows from Lemmas 2 and 3:

Theorem 2. *For deterministic u_j, SRTS is θ_1-competitive for $\mathcal{P}$, where*

$$\theta_1 = \min\left\{4 + \frac{\beta_{max} - 2}{m+1}, m+3\right\}. \tag{14}$$

From Theorem 2 it can be observed that SRTS yields a competitive ratio with some interesting features. First, unlike in the case of SO-Shmoys, θ_1 is independent of n. This is important, since the number of tasks in common applications such as cloud computing can be large.

Second, if $\beta_{\max}$ is independent of m, then a simple upper bound for θ_1 in terms of $\beta_{\max}$ can be obtained by solving for m in the following equation:

$$4 + \frac{\beta_{\max} - 2}{m+1} = m + 3.$$

The solution is given by $m = \sqrt{\beta_{\max} - 1}$. Substituting this value in (14) and noting that $m \geq 1$, we obtain

$$\theta_1 \leq \begin{cases} 4, & 0 < \beta_{\max} < 2 \\ \sqrt{\beta_{\max} - 1} + 3, & \beta_{\max} \geq 2. \end{cases}$$

Therefore, in this case SRTS has constant competitive ratio independent of m.

Third, consider the case where $\beta_{\max}$ is a function of m. As an example, this may happen if we assume that the capacity of the remote processor is always at a similar level as the combined capacity of all m local processors. From (14), we observe that as long as $\beta_{\max}$ is $O(m)$, θ_1 is $O(1)$. In other words, if the unknown processing speed is $O(m)$ times the processing speed of each known processor, SRTS has asymptotically constant competitive ratio. Note that in most practical parallel computing systems, the speed difference between the unknown (e.g., cloud) processor and a known (e.g., local) processor is not excessive. Therefore, in this case we expect SRTS to have asymptotically constant competitive ratio in general.

5.2 Worst Case Bound for Random u_j

In the case of random u_j, if a task is restarted on processor $m+1$ under SRTS, it acquires a different processing time. Therefore, $C^*_{\max}$ and $C_{\max}(\mathbf{s}^{\text{SRTS}})$ are not directly comparable. Nevertheless, we may compare their expected values over the random realizations of u_j. Here, we observe that Theorem 2 can be generalized to the competitive ratio definition in (2).

Theorem 3. *For random u_j, SRTS has the following upper bound for the expected makespan ratio:*

$$\frac{\mathbb{E}[C_{max}(\mathbf{s}^{SRTS}(P,\{u_j^{(1)}\},\{u_j^{(2)}\}))]}{\mathbb{E}[C_{max}(\mathbf{s}^*(P,\{u_j\})]} \leq \theta_2,$$

where

$$\theta_2 = \min\left\{4 + \frac{\max_j(\frac{a_j}{\nu_{min}}) - 2}{m+1}, m+3\right\}$$

and $\nu_{min} > 0$ is the minimum value in the sample space from which u_j are drawn.

Proof. Recall that when a task is restarted on processor $m+1$ under SRTS, it acquires a different processing time. For a given $\{P, F_u\}$, let $\{\nu_j^{(1)}\}$ denote the realization of $\{u_j^{(1)}\}$ and $\{\nu_j^{(2)}\}$ denote the realization of $\{u_j^{(2)}\}$. We observe an important fact that the realization of $\{u_j\}$ experienced in the second iteration of SRTS does not appear in the proofs of Lemmas 1, 2, 3 and Theorem 2. Since Theorem 2 holds for the deterministic problem instance P with processing times $\{\nu_j^{(1)}\}$, using the above observation, we obtain

$$\begin{aligned}
&C_{\max}(\mathbf{s}^{\text{SRTS}}(P,\{\nu_j^{(1)}\},\{\nu_j^{(2)}\})) \\
&\leq \min\left\{4 + \frac{\max_j(\frac{a_j}{\nu_j^{(1)}}) - 2}{m+1}, m+3\right\} C_{\max}(\mathbf{s}^*(P,\{\nu_j^{(1)}\})) \\
&\leq \theta_2 C_{\max}(\mathbf{s}^*(P,\{\nu_j^{(1)}\})), \qquad (15)
\end{aligned}$$

where we have used $\max_j(a_j/\nu_j^{(1)}) \leq \max_j(a_j/\nu_{\min})$, for all $\{\nu_j^{(1)}\}$ and for all j. We note that the inequality in (15) is valid for any realizations $\{\nu_j^{(1)}\}$ and $\{\nu_j^{(2)}\}$. Therefore, we have

$$\begin{aligned}
&C_{\max}(\mathbf{s}^{\text{SRTS}}(P,\{u_j^{(1)}\},\{u_j^{(2)}\})) \leq \theta_2 C_{\max}(\mathbf{s}^*(P,\{u_j^{(1)}\})) \\
\Rightarrow &\mathbb{E}[C_{\max}(\mathbf{s}^{\text{SRTS}}(P,\{u_j^{(1)}\},\{u_j^{(2)}\}))] \leq \theta_2 \mathbb{E}[C_{\max}(\mathbf{s}^*(P,\{u_j\}))]
\end{aligned}$$

For the second step above, we have taken expectations with respect to the joint distribution of $\{u_j^{(1)}\}$ and $\{u_j^{(2)}\}$ on both sides and used the fact that $\{u_j^{(1)}\}$ and $\{u_j\}$ have the same distribution F_u. □

Remark: We note that the result in Theorem 3 is valid for generally distributed task processing times $\{u_j\}$. More interestingly, it is valid for any distribution for $\{u_j^{(2)}\}$, and $\{u_j^{(2)}\}$ and $\{u_j^{(1)}\}$ can be correlated; for example for deterministic $\{u_j\}$ both realizations are the same. This result stems from the fact that, after the first iteration of SRTS, using a simple schedule where all the restarted tasks are scheduled locally still results in (15). Thus, the processing times of tasks in the second iteration does not effect the result.

The competitive ratio θ_2 is loose when compared with θ_1 as it constitutes the term $\max_j(\frac{a_j}{\nu_{\min}})$. Nevertheless, θ_2 has the same properties of θ_1, namely, θ_2 is independent of n, is constant if $\max_j(\frac{a_j}{\nu_{\min}})$ is independent of m, and is asymptotically constant if $\max_j(\frac{a_j}{\nu_{\min}})$ is dependent of m.

6 Extensions

In this section we present three important extensions to SRTS. First, we extend the competitive ratio results proved in Section 5 for the case of uniform known processors. Second, we present a method to extend SRTS for tasks that arrive dynamically in time, whose arrival times may not be known a priori. Third, using the ideas of SRTS we propose SRTS-M for the case where there are multiple remote processors.

6.1 Uniform Known Processors

Under the uniform known processors model, the processing time of a task j on a processor i is given by $\rho_i a_j$, where ρ_i is the slow-down factor of processor i. Our motivation for considering this extension is that it allows us to model systems with heterogeneous processors, which are quite prevalent today. The following are potential use cases we can model using this extension: 1) ARM big.LITTLE CPU chip in a mobile device where different processor cores have different speeds; 2) a hybrid cloud system, where a small scale enterprises owns a set of local processors (heterogeneous) and also subscribes to a public cloud for more computing power; and 3) a group of mobile devices in proximity collaborate to perform computation and may enlist the help of an edge device for offloading. In the above use cases, using the model of uniform parallel processors for the local processors is more accurate.

Without loss of generality, we consider $\rho_1 = 1$, and $\rho_1 \leq \rho_2 \leq \ldots \leq \rho_m$, where $\rho_{\max} = \rho_m$. This implies that processor 1 is the fastest and processor m is the slowest.

Theorem 4. *If the known processors are uniform and the slowest processor is ρ_{max} times slower than the fastest processor, then SRTS is $\rho_{max}\theta_1$-competitive for $\mathcal{P}$.*

Proof. To prove the result we consider a hypothetical system of $m+1$ processors, where m processors are identical to processor 1 and the $(m+1)$-th processor is identical to the remote processor in the original system. Note that any feasible schedule for the original system is a feasible

schedule for the hypothetical system and vice-versa. Given a schedule $\mathbf{s}$, we use $C^I_{\max}(\mathbf{s})$ to denote the resulting makespan in the hypothetical system.

For any schedule $\mathbf{s}$, we claim that $C_{\max}(\mathbf{s}) \leq \rho_{\max} C^I_{\max}(\mathbf{s})$, where $C_{\max}(\mathbf{s})$ is the makespan in the original system. To see this, note that $\rho_i \leq \rho_{\max}$, for all $i \in \mathcal{Q}$ and the same schedule is used to schedule tasks both in the hypothetical and the original systems. This implies the completion times on the processors in the original system cannot be greater than $\rho_{\max}$ times the completion times on the processors in the hypothetical system. Using the above claim, we obtain

$$C_{\max}(\mathbf{s}^{\text{SRTS}}) \leq \rho_{\max} C^I_{\max}(\mathbf{s}^{\text{SRTS}}). \tag{16}$$

Let $\mathbf{s}^{*I}$ denote the optimal schedule for the hypothetical system. Using the result in Theorem 2 for the schedules in the hypothetical system, we obtain

$$C^I_{\max}(\mathbf{s}^{\text{SRTS}}) \leq \theta_1 C^I_{\max}(\mathbf{s}^{*I}). \tag{17}$$

Also, we have

$$C^I_{\max}(\mathbf{s}^{*I}) \leq C^I_{\max}(\mathbf{s}^*), \tag{18}$$

where $C^I_{\max}(\mathbf{s}^*)$ is the makespan in the hypothetical system when $\mathbf{s}^*$ is used. By the construction of the hypothetical system we have

$$C^I_{\max}(\mathbf{s}^*) \leq C_{\max}(\mathbf{s}^*). \tag{19}$$

The result follows from (16), (17) (18) and (19). □

Theorem 5. *For random u_j, if the known processors are uniform and the slowest processor is ρ_{max} times slower than the fastest processor, then SRTS has the following upper bound for the expected makespan ratio:*

$$\frac{\mathbb{E}[C_{max}(\mathbf{s}(P,\{u_j\}))]}{\mathbb{E}[C_{max}(\mathbf{s}^*(P,\{u_j\})]} \leq \rho_{max}\theta_2.$$

Proof. The proof is similar to the proof of Theorem 4 and is omitted. □

6.2 Dynamic Task Arrivals

In $\mathcal{P}$, we have assumed that all tasks arrive at time zero. However, in general, tasks may arrive dynamically in time, and their arrival times may not be known a priori. In this case, we denote the generalized problem by $\mathcal{P}_r$. In the offline setting, the arrival times of all the tasks and their processing times on all processors are known at time zero, and we use $C^{r*}_{\max}$ to denote the minimum makespan.

Again, we first focus on the case where u_j are deterministic. Given a θ-competitive online or semi-online algorithm for solving $\mathcal{P}$, an algorithm that provides 2θ competitive ratio for $\mathcal{P}_r$ was proposed in [15]. In [36], Sgall pointed out that if we know the release time of the last task, then accumulating the tasks till the last task arrival and then scheduling all tasks using the θ-competitive algorithm will result in a schedule with competitive ratio $\theta+1$. However, it can be noted that waiting till the last task arrival to schedule the tasks is inefficient. Instead, we propose an algorithm that does not wait till the last task arrival to schedule the tasks and still achieves $\theta + 1$ competitive ratio for $\mathcal{P}_r$.

Given that there is an indication that a task is the last task when it arrives, our algorithm is obtained by modifying the algorithm proposed in [15]. As in [15], a θ-competitive algorithm is used for scheduling the tasks that arrive at time zero. The tasks that are arriving are accumulated till the time at which all the tasks scheduled at time zero are finished. All the accumulated tasks are then schedule using the θ-competitive algorithm and the procedure is repeated. In contrast to [15], when the last task arrives all tasks that are under processing are cancelled. The cancelled tasks and the last task are scheduled using the θ-competitive algorithm.

We present our algorithm for solving $\mathcal{P}_r$ in Algorithm 2. It uses the following definitions. Let A_θ be a θ-competitive algorithm for solving $\mathcal{P}$. Let $\mathcal{B}_t$ be the set of tasks available but not yet scheduled at time t, and τ_t be the resultant schedule length when tasks from $\mathcal{B}_t$ are scheduled by A_θ. Without loss of generality, we assume the first task arrive at time $t = 0$.

Algorithm 2: General algorithm for solving $\mathcal{P}_r$

1: At $t = 0$, schedule $\mathcal{B}_0$ using algorithm A_θ and observe schedule length τ_0.
2: **repeat**
3: Wait until time $t + \tau_t$ or the last task arrival, whichever happens first
4: **if** the last task has arrived **then**
5: Cancel any task under execution
6: Schedule all unfinished tasks using A_θ
7: Exit
8: **end if**
9: Schedule tasks from $\mathcal{B}_t$ using A_θ.
10: **until** all n tasks are finished

Denote by $\mathbf{s}^R$ the schedule given by Algorithm 2. The following proposition expresses its competitive ratio.

Lemma 4. $C_{max}(\mathbf{s}^R) \leq (\theta+1)C^{r*}_{max}$

Proof. Let t_n denote the arrival time of the last task n, and C^* be the optimal makespan if all tasks were available at time zero. We know that $\mathcal{B}_{t_n}$ is the set of all unfinished tasks from $\mathcal{T}$ at time t_n. When tasks from $\mathcal{B}_{t_n}$ are scheduled using A_θ, the resultant schedule length τ_{t_n} cannot be greater than θC^*, because $\mathcal{B}_{t_n} \subseteq \mathcal{T}$ and θC^* is the upper bound for the makespan produced by A_θ for scheduling tasks from $\mathcal{T}$ under the assumption that all of them are available at time zero. Therefore, we have

$$C_{\max}(\mathbf{s}^R) = \tau_{t_n} + t_n \leq (\theta+1)C^{r*}_{\max},$$

where we have used the fact that $C^{r*}_{\max}$ cannot be smaller than the arrival time of the last task and $C^{r*}_{\max} \geq C^*$. □

When SRTS is used as A_θ, we refer to Algorithm 2 as Dynamic SRTS (DSRTS). From Theorems 2 and 4 and Lemma 4, we immediately arrive at the following result.

Theorem 6. *DSRTS is (θ_1+1)-competitive for $\mathcal{P}_r$, where θ_1 is given by* (14)*. If the known processors are uniform, then DSRTS is $(\rho_{max}\theta_1+1)$-competitive for $\mathcal{P}_r$.*

For the case of random u_j, a performance bound for DSRTS is presented in the following theorem.

Theorem 7. *Let t_j denote the arrival time of job j and $\{P_r, \{u_j\}\}$ be a problem instance of $\mathcal{P}_r$, where $P_r = \{m, n, \{a_j\}, \{t_j\}\}$. For random u_j, DSRTS has the following upper bound for the expected makespan ratio:*

$$\frac{\mathbb{E}[C_{max}(\mathbf{s}(P_r, \{u_j\}))]}{\mathbb{E}[C_{max}(\mathbf{s}^*(P_r, \{u_j\})]} \leq \theta_2 + 1.$$

Proof. The result can be obtained by applying the same arguments from the proof of Lemma 4 and Theorem 3 for a single realization of $\{u_j\}$ and then taking expectation. The proof is omitted. □

6.3 Multiple Unknown Processors

In this section, we consider the problem where multiple remote processors with unknown processing times are available for offloading the computational tasks. We consider the general case where the remote processors are non-identical and index them by $i \in \mathcal{Q}' = \{m+1, \ldots, m+m'\}$. Recall that Shmoys' algorithm can be applied to this case and is $O(\log n)$-competitive if the processing times are deterministic. However, as noted before it has very poor average performance. Instead, learning from the proven ideas of SRTS, we propose a heuristic SRTS-Multiple (SRTS-M) algorithm to solve this problem.

Similar to SRTS, SRTS-M also has two iterations. The tasks are listed in the ascending order of a_j values. Without loss of generality, consider $a_1 \leq a_2 \leq \ldots \leq a_n$. In the first iteration of SRTS-M, whenever a known processor becomes idle, it is given a task from the *start* of the list. Similarly, whenever a remote processor becomes idle it is given a task from the *end* of the list. A task j_1 that is scheduled on a remote processor is cancelled in the first iteration if its processing on the remote processors exceeds the estimation time $\sum_{j=j_1}^{n} a_j$. The rationale behind this choice of the estimation times is the following. Consider a hypothetical powerful single remote processor in place of the set of remote processors, and we use SRTS to schedule the tasks. In this case, in the first iteration of SRTS, the time that any offloaded task j_1 is completed or cancelled is upper bounded by $\sum_{j=j_1}^{n} a_j$. We note that our choice of estimation time $\sum_{j=j_1}^{n} a_j$ for any offloaded task j_1 in SRTS-M is greater than or equal to the estimation time a_{j_1} used in SRTS. This higher estimation time in SRTS-M potentially avoids unnecessary restarts on multiple remote processors.

The details of SRTS-M are presented in Algorithm 3. Similarly to SRTS, SRTS-M runs in $O(n \log n)$ time and can be readily implemented in practice by a local scheduler. However, it is challenging to derive its competitive ratio, because restarting an offloaded task on an unknown processor does not reveal any information about its processing time on another unknown processor, thereby making it difficult to derive an upper bound expression for the makespan. Instead, in Section 7, we show using simulation that it significantly out performs the best existing alternatives. Further, when SRTS-M is used as A_θ in Algorithm 2, we call it Dynamic SRTS-M (DSRTS-M) and study its average performance in Section 7.

Algorithm 3: SRTS-M

$\mathcal{T}^{(1)} = \mathcal{T}$
for $l = 1$ to 2 **do**
  Sort $\mathcal{T}^{(l)}$ in the ascending order of a_j. WLOG, re-index tasks such that $a_1 \leq a_2 \leq \ldots \leq a_{|\mathcal{T}^{(l)}|}$.
  $j_1 = |\mathcal{T}^{(l)}| + 1$, $j_0 = 0$
  for $k = m+1$ to $m + \min\{m', |\mathcal{T}^{(l)}|\}$ **do**
    $j_1 = j_1 - 1$
    Start processing task j_1 on processor k
    if $l = 1$ **then**
      Cancel task j_1 if its execution time exceeds $\sum_{j=j_1}^{n} a_{j_1}$ and include it in $\mathcal{T}^{(l+1)}$
    end if
  end for
  for $k = 1$ to $\min\{m, |\mathcal{T}^{(l)}|\}$ **do**
    $j_0 = j_0 + 1$
    Start processing task j_0 on processor k.
  end for
  while $\mathcal{T}^{(l)} \neq \emptyset$ **do**
    Wait until next event E occurs
    if $E =$ a processor $\hat{i} \in \mathcal{Q}$ becomes idle **then**
      Let task j be the last task completed on $\hat{i}$
      Cancel task j if it is scheduled on some processor from $\mathcal{Q}'$
      $\mathcal{T}^{(l)} = \mathcal{T}^{(l)} \backslash \{j\}$
      $j_0 = j_0 + 1$
      If task j_0 is not completed or cancelled yet, schedule it on processor $\hat{i}$
    else if $E =$ a processor $\hat{i} \in \mathcal{Q}'$ becomes idle **then**
      Cancel task j_1 if it is scheduled on some processor from $\mathcal{Q}$
      $\mathcal{T}^{(l)} = \mathcal{T}^{(l)} \backslash \{j_1\}$
      $j_1 = j_1 - 1$
      If task j_1 is not completed yet, schedule it on processor $\hat{i}$
      if $l = 1$ **then**
        Cancel task j_1 if its execution time exceeds $\sum_{j=j_1}^{n} a_{j_1}$ and include it in $\mathcal{T}^{(l+1)}$
      end if
    end if
  end while
end for

7 Evaluation of Average Performance

In addition to the competitive ratios derived for SRTS in Section 5, we are interested in studying the average performance of SRTS and SRTS-M over general parameter values. Toward this end, we conduct simulation in MATLAB for evaluation and comparison with several well-known alternatives.

7.1 Single Remote Processor

We compare SRTS with LS, SO-LPT, and SO-Shmoys. In addition, we also consider a *Semi-Online Shortest Processing Time* (SO-SPT) algorithm, which is the same as SO-LPT except that the known process times a_j are listed in ascending order. For Figures 2 and 3, a_j and u_j are generated independently from an exponential distribution. We set the

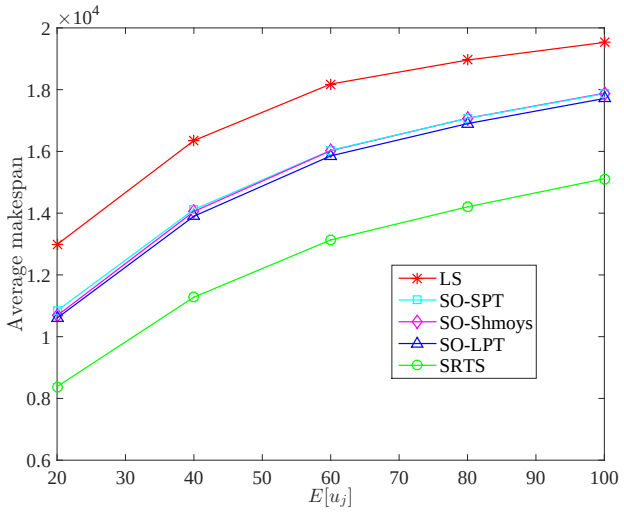

Fig. 2. Effect of varying $\mathbb{E}[u_j]$. Single remote processor.

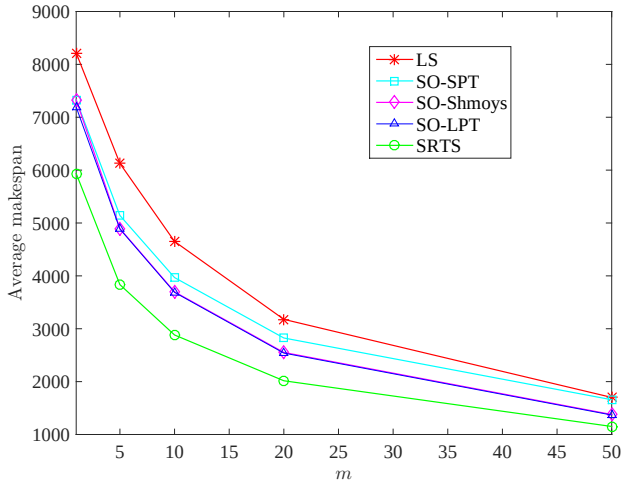

Fig. 3. Effect of the number of local processors. Single remote processor.

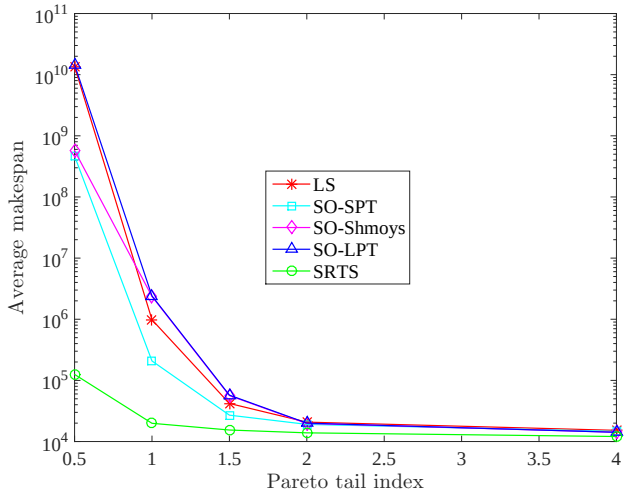

Fig. 4. Effect of the tail of distribution, for $m = 4$. Single remote processor.

number of tasks $n = 1500$. For each data point, we average the makespan over $10,000$ runs, combining 100 realizations each for $\{a_j\}$ and $\{u_j\}$. In Figure 2, we set $m = 10$, $E[a_j] = 60$, and vary $E[u_j]$. In Figure 3, we set $E[a_j] = 60$, $E[u_j] = 6$, and vary m. We choose $E[a_j]$ larger than $E[u_j]$ to reflect the practical scenario where the remote server is often faster than the local processors. We observe that SRTS outperforms all other algorithms. It provides a makespan reduction up to 30% compared with the best alternatives of SO-LPT and SO-Shmoys.

Similar performance trends have been observed when we use other distributions. In general, the performance advantage of SRTS is more pronounced when the distribution of a_j and u_j has a heavier tail. This is because a heavier tail implies more chances for some extremely long processing times, which can clog an unknown processor in algorithms with deterministic scheduling order, such as LS and SO-LPT, and lead to high inefficiency in algorithms with multiple restarts and no simultaneous processing, such as SO-Shmoys. This is illustrated in Figure 4, where we generate $\{a_j\}$ and $\{u_j\}$ using the Pareto distribution. The Pareto scale parameters of $\{a_j\}$ and $\{u_j\}$ are given by the Pareto tail index multiplied by 60 and 6, respectively. We note that as the Pareto tail index parameter increases, the *heaviness* of the tail decreases.

7.2 Uniform Local Processors and Multiple Remote Processors

In this subsection we consider the system of uniform local processors and multiple remote processors and study the average performance of SRTS-M. The processing times a_j and u_j are generated independently from exponential distributions. The slow down factors ρ_i are chosen uniformly from the set $\{1, 2, \ldots, 20\}$. The default parameters are $m = 4$, $m' = 4$, $\mathbb{E}[a_j] = 60$, and $\mathbb{E}[u_j] = 60$. In Figures 5 and 6 we compare the average performance of SRTS-M with the alternate algorithms mentioned in Section 7.1. We observe that SRTS-M provides $20 - 30\%$ reduction in the average makespan over a wide range of $\mathbb{E}[u_j]$ and m' values.

For task processing times generated using *heavy-tailed* distributions as in the previous section, we observe that SRTS-M significantly reduces the makespan when compared with the alternatives. This is illustrated in Figure 7. We also note that the performance of SO-Shmoys degrades significantly with multiple unknown processors because of the multiple rounds of restarts on all the unknown processors.

7.3 Dynamic Task Arrivals

We study the scenario of dynamic task arrivals for the system of uniform local processors and multiple remote processors. For dynamic task arrivals, the processing time of a task in on a local processor is not known until it arrives. Therefore, except LS, other alternatives SO-LPT, SO-SPT and SO-Shmoys cannot be directly used for this scenario. For Figures 8 and 9, the same parameter values are used from Section 7.2, except we simulate for the makespan of 10^5 task arrivals for each data point, with two different arrival processes. The first is a single-task arrival process with inter-arrival times between tasks chosen from an exponential

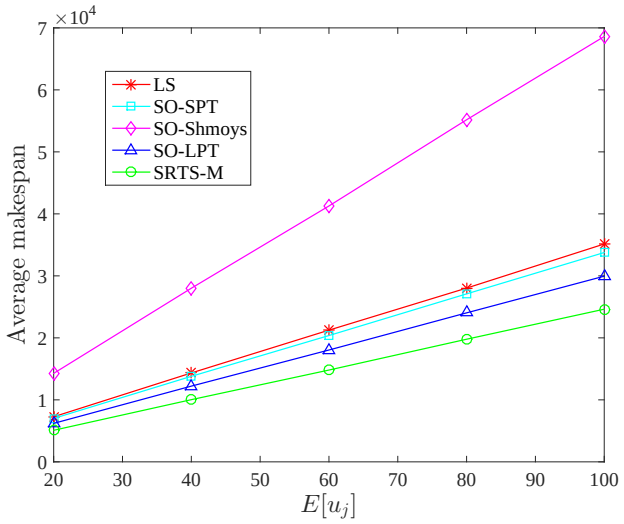

Fig. 5. Effect of varying $\mathbb{E}[u_j]$. Multiple remote and uniform local processors.

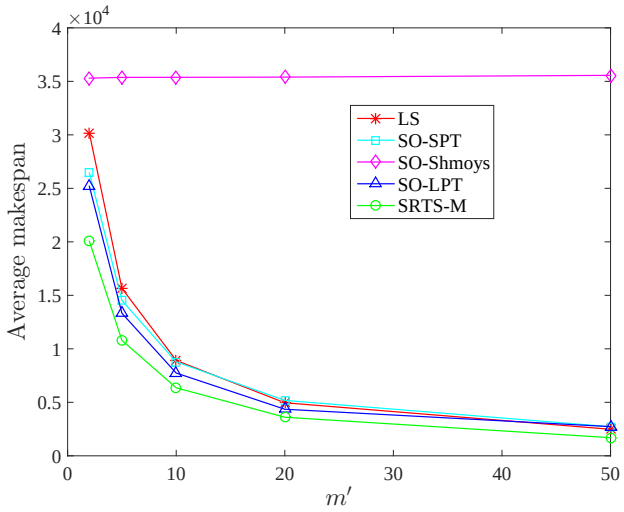

Fig. 6. Effect of the number of remote processors. Uniform local processors.

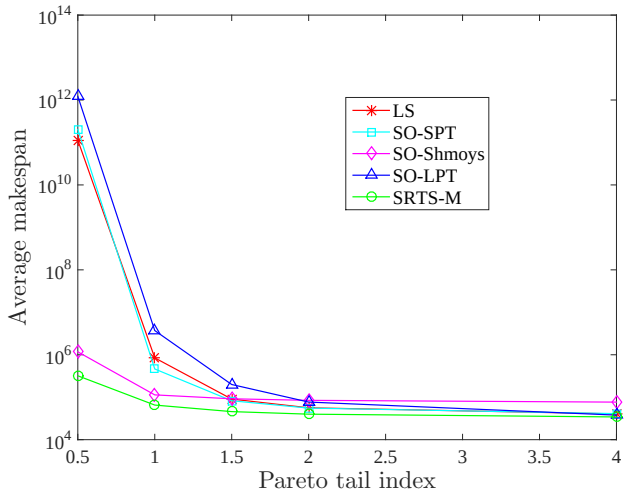

Fig. 7. Effect of the tail of distribution, for $m = 4$, and $m' = 4$. Uniform local processors.

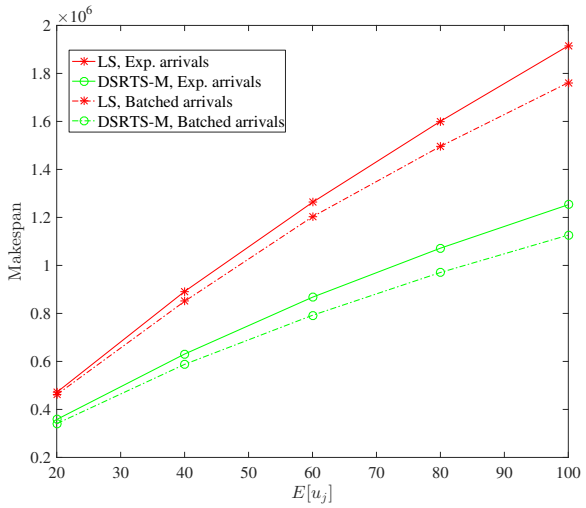

Fig. 8. Effect of varying $\mathbf{E}[u_j]$, for dynamic task arrivals. Multiple remote and uniform local processors

distribution with mean 0.1, and the second is a batched-periodic-arrival process in which inter-arrival time between batches is set to 1 and the number of arrivals per batch is chosen uniformly from the set $\{1, 2, \ldots, 10\}$. These two processes are labelled as Exp. arrivals and Batched arrivals, respectively, in the figures. We observe that compared to LS, DSRTS-M provides a reduction in makespan of up to 13% for the case of varying mean processing times, and up to 45% for the case of varying number of remote processors.

8 Conclusion

We have proposed SRTS for semi-online scheduling of n tasks on m known (or local) processors and one unknown (or remote) processor, aiming to reduce the makespan of processing all tasks. If the unknown task processing times are deterministic, the competitive ratio of SRTS is shown to always be constant when the processing times are independent of m, and asymptotically constant in practice when the processing times are dependent on m. We derive a similar result for the case where the unknown task processing times are random. We have extended SRTS for the case where tasks arrive dynamically over time and proved a competitive ratio that is one more than the competitive ratio of SRTS. We have also extended SRTS for the case of multiple unknown processors and proposed SRTS-M. Our simulation results show that SRTS and SRTS-M provide substantial performance improvement over existing alternatives in terms of the average makespan, and the performance improvement is more pronounced if the task processing times follow heavy-tailed distributions.

References

[1] J. P. Champati and B. Liang, "Single restart with time stamps for computational offloading in a semi-online setting," in *Proc. IEEE INFOCOM*, 2017.

[2] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 84–106, Jan 2013.

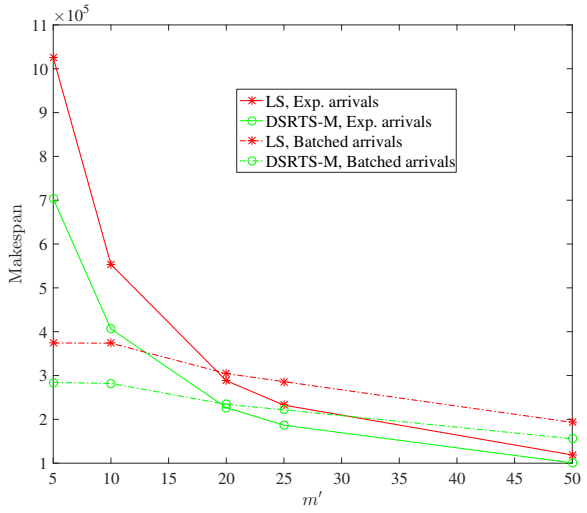

Fig. 9. Effect of the number of remote processors, for dynamic task arrivals. Multiple remote and uniform local processors

[3] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mob. Netw. Appl.*, vol. 18, no. 1, pp. 129–140, Feb. 2013.

[4] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing - a key technology towards 5g," European Telecommunications Standards Institute (ETSI) White Paper, 2015.

[5] "Mobile edge computing (MEC); framework and reference architecture," ETSI Group Specification MEC 003 V1.1.1, 2016.

[6] B. Liang, *"Mobile edge computing," in Key Technologies for 5G Wireless Systems*, V. W. S. Wong, R. Schober, D. W. K. Ng, and L.-C. Wang, Eds. Cambridge University Press, 2017.

[7] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, Aug. 2017.

[8] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct. 2009.

[9] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *Proc. ACM SIGOPS European Workshop*, 2002, pp. 87–92.

[10] M. Conti and M. Kumar, "Opportunities in opportunistic computing," *Computer*, vol. 43, no. 1, pp. 42–50, Jan. 2010.

[11] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," in *Proc. MobiHoc*, 2012, pp. 145–154.

[12] M. Pitkänen, T. Kärkkäinen, J. Ott, M. Conti, A. Passarella, S. Giordano, D. Puccinelli, F. Legendre, S. Trifunovic, K. Hummel, M. May, N. Hegde, and T. Spyropoulos, "Scampi: Service platform for social aware mobile and pervasive computing," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 503–508, Sep. 2012.

[13] M. Drozdowski, *Scheduling for Parallel Processing*. Springer Publishing Company, 2009.

[14] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, pp. 1563–1541, 1966.

[15] D. B. Shmoys, J. Wein, and D. P. Williamson, "Scheduling parallel machines on-line," *SIAM J. Comput.*, vol. 24, no. 6, pp. 1313–1331, Dec. 1995.

[16] J. P. Champati and B. Liang, "One-restart algorithm for scheduling and offloading in a hybrid cloud," in *Proc. IEEE/ACM IWQoS*, Jun. 2015.

[17] J. P. Champati and B. Liang, "Delay and cost optimization in computational offloading systems with unknown task processing times," to appear in *IEEE Transactions on Cloud Computing*, 2019, DOI: 10.1109/TCC.2019.2924634.

[18] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, pp. 416–429, 1969.

[19] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of discrete mathematics*, vol. 5, no. 2, pp. 287–326, 1979.

[20] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*, 1st ed. New York, NY, USA: Cambridge University Press, 2011.

[21] C. N. Potts, "Analysis of a linear programming heuristic for scheduling unrelated parallel machines," *Discrete Applied Mathematics*, vol. 10, no. 2, pp. 155–164, 1985.

[22] J. K. Lenstra, D. B. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Math. Program.*, vol. 46, no. 3, pp. 259–271, Feb. 1990.

[23] H. Kellerer, V. Kotov, M. G. Speranza, and Z. Tuza, "Semi on-line algorithms for the partition problem," *Operations Research Letters*, vol. 21, no. 5, pp. 235 – 242, 1997.

[24] T. E. Cheng, H. Kellerer, and V. Kotov, "Semi-on-line multiprocessor scheduling with given total processing time," *Theoretical Computer Science*, vol. 337, no. 13, pp. 134 – 146, 2005.

[25] C. Ng, Z. Tan, Y. He, and T. Cheng, "Two semi-online scheduling problems on two uniform machines," *Theoretical Computer Science*, vol. 410, no. 810, pp. 776 – 792, 2009.

[26] S. Albers and M. Hellwig, "Semi-online scheduling revisited," *Theoretical Computer Science*, vol. 443, no. 0, pp. 1 – 9, 2012.

[27] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proc. ACM MobiSys*, 2010, pp. 49–62.

[28] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, 2012, pp. 945–953.

[29] J. Champati and B. Liang, "Energy compensated cloud assistance in mobile cloud computing," in *Proc. IEEE INFOCOM Workshop on Mobile Cloud Computing*, April 2014.

[30] Y. Cui, J. Song, K. Ren, M. Li, Z. Li, Q. Ren, and Y. Zhang, "Software defined cooperative offloading for mobile cloudlets," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1746–1760, June 2017.

[31] M. Shifrin, R. Atar, and I. Cidon, "Optimal scheduling in the hybrid-cloud," in *Proc. IFIP/IEEE International Symposium on Integrated Network Management*, 2013, pp. 51–59.

[32] A. Pasdar, K. Almiani, and Y. C. Lee, "Data-aware scheduling of scientific workflows in hybrid clouds," in *Proc. International Conference on Computational Science*, 2018, pp. 708–714.

[33] R. D. Friese, M. Halappanavar, A. V. Sathanur, M. Schram, D. J. Kerbyson, and L. de la Torre, "Towards efficient resource allocation for distributed workflows under demand uncertainties," in *Job Scheduling Strategies for Parallel Processing, Springer*, 2018, pp. 103–121.

[34] T. H. Bhuiyan, M. Halappanavar, R. D. Friese, H. Medal, L. de la Torre, A. Sathanur, and N. R. Tallent, "Stochastic programming approach for resource selection under demand uncertainty," in *Job Scheduling Strategies for Parallel Processing*, 2019, pp. 107–126.

[35] T. Gonzalez, O. H. Ibarra, and S. Sahni, "Bounds for LPT Schedules on Uniform Processors," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 155–166, 1977.

[36] J. Sgall, *Online Algorithms: The State of the Art*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, ch. On-line scheduling, pp. 196–231.

Jaya Prakash Champati is a post-doctoral researcher at Information Science and Engineering Department, KTH Royal Institute of Technology. He obtained his PhD degree from Electrical and Computer Engineering Department at University of Toronto. He obtained his bachelor of technology degree from National Institute of Technology Warangal, India, and master of technology degree from Indian Institute of Technology (IIT) Bombay, India. His general research interest is in the design and analysis of algorithms for scheduling problems that arise in networking and information systems. His favorite tools are combinatorial optimization, approximations algorithms, network calculus, and queueing theory. Prior to joining PhD he worked at Broadcom Communications, where he was part of protocol stack development team for an upcoming LTE chipset project. He was a recipient of the best paper award at National Conference on Communications 2011, IISc, Bangalore, India.

Ben Liang received honors-simultaneous B.Sc. (valedictorian) and M.Sc. degrees in Electrical Engineering from Polytechnic University in Brooklyn, New York, in 1997 and the Ph.D. degree in Electrical Engineering with a minor in Computer Science from Cornell University in Ithaca, New York, in 2001. In the 2001 - 2002 academic year, he was a visiting lecturer and post-doctoral research associate at Cornell University. He joined the Department of Electrical and Computer Engineering at the University of Toronto in 2002, where he is now a Professor. His current research interests are in networked systems and mobile communications. He has served on the editorial boards of the IEEE Transactions on Mobile Computing since 2017 and the IEEE Transactions on Communications since 2014, and he was an editor for the IEEE Transactions on Wireless Communications from 2008 to 2013 and an associate editor for Wiley Security and Communication Networks from 2007 to 2016. He regularly serves on the organizational and technical committees of a number of conferences. He is a Fellow of IEEE and a member of ACM and Tau Beta Pi.