Optimizing the stretch of independent tasks on a cluster: From sequential tasks to moldable tasks

Erik Saule[a,*], Doruk Bozdağ[a], Ümit V. Çatalyürek[a,b]

[a] *Department of Biomedical Informatics, The Ohio State University, Columbus OH, USA*
[b] *Department of Electrical and Computer Engineering, The Ohio State University, Columbus OH, USA*

Abstract

This paper addresses the problem of scheduling non-preemptive moldable tasks to minimize the stretch of the tasks in an online non-clairvoyant setting. To the best of the authors' knowledge, this problem has never been studied before. To tackle this problem, first the sequential sub-problem is studied through the lens of the approximation theory. An algorithm, called DASEDF, is proposed and, through simulations, it is shown to outperform the First-Come, First-Served scheme. Furthermore, it is observed that the machine availability is a key to getting good stretch values. Then, the moldable task scheduling problem is considered and by leveraging the results from the sequential case, another algorithm, DBOS, is proposed to optimize the stretch while scheduling moldable tasks. This work is motivated by a task scheduling problem in the context of parallel short sequence mapping which has important applications in biology and genetics. The proposed DBOS algorithm is evaluated both on synthetic data sets that represent short sequence mapping requests and on data sets generated using log files of real production clusters. The results show that the DBOS algorithm significantly outperforms the two state-of-the-art task scheduling algorithms on stretch optimization.

Keywords: Online Scheduling, Maximum Stretch, Approximation Algorithm, Resource Augmentation, Simulation, Job Scheduling, Sequential Task, Moldable Task

*Corresponding author
Email addresses: esaule@bmi.osu.edu (Erik Saule), bozdag.1@osu.edu (Doruk Bozdağ), umit@bmi.osu.edu (Ümit V. Çatalyürek)

1. Introduction

In a multi-user cluster environment, allocation of sufficient computing resources to each submitted task is orchestrated by the scheduler component of the cluster's resource manager (such as MAUI [1] or OAR [2]). Schedulers are typically designed to ensure low response time, high cluster utilization, fairness between tasks and users, and starvation avoidence. A common measure used in optimization of these factors is the *flow-time* (also called turnaround time), defined as the time elapsed between a task's arrival to a system and the completion of its execution. Flow-times are aggregated into single performance indices, such as average flow-time, maximum flow-time, or a per user metric such as the average flow-time per user (which would need to be aggregated by a fairness function, such as a maximum).

The major drawback of flow-time-based performance indices is that they are oblivious to task sizes. Since the time spent in the system is the only important factor, a 10-minute task and a 10-hour task that stay in the system for the same amount of time contribute equally to a flow-time-based objective function. (For instance, if these two tasks are released at the same time and only one processor is available, the second task to be executed gets a flow-time of 10 hours plus 10 minutes, regardless of the order in which they are executed.) This results in unfair schedules against smaller tasks in terms of the delay they experience relative to their size. To address this problem, the *stretch* performance index has been introduced in [3]. The stretch of a task is defined as the ratio of the task's flow-time to its size. By including the size of the tasks in its definition, stretch promotes fairness in the system.

In this work, we study the problem of optimizing the stretch of *moldable* tasks in a cluster computing environment. A parallel task is said to be moldable[1] if the number of processors to be used for executing the task can be decided by the scheduler at runtime (usually based on instantaneous load and availability in the system). This is in contrast to *rigid* tasks, for which the number of processors to be used is provided by the user at submission time.

[1]They were originally called malleable tasks [4]. Feitelson *et al.* [5] made the distinction between moldable tasks which run on a constant number of processors decided at runtime and malleable tasks which can accomodate variation in the number of processors at runtime. Although they were still called malleable in more recent works, this denomination is widely accepted.

(Notice that *sequential* tasks are a special case of rigid tasks that run on a single processor). To the best of our knowledge, stretch has never been considered as the optimization criteria for scheduling moldable tasks.

This work is motivated by the parallel short sequence mapping problem [6], which has important applications in biology and genetics including whole-genome resequencing, transcriptome analysis and ChIP sequencing. Next-generation sequencing instruments are capable of sequencing hundreds of millions of short DNA/RNA sequences in a single run. Mapping those sequences to a reference genome sequence is the most computation-intensive part of many analysis workflows and it takes about a day on a modern desktop. Provided that the sequencing instruments will soon be capable of sequencing the entire human genome in 15 minutes [7], it is extremely important to speedup the mapping process to effectively benefit from faster sequencing rates. In a recent work [6], we have proposed parallelization techniques for short sequence mapping problem to address this issue. However, since providing a computer cluster for every sequencing instrument is not cost efficient, solutions that pool various sequence mapping requests on a single cluster are needed. Short sequence mapping jobs suit the moldable tasks model, since they can run on different number of processors and the execution times on a given number of processor can be accurately predicted [6]. Furthermore, their execution times can vary several orders of magnitude depending on the number of short sequences to be mapped and the size of the reference genome. These properties give rise to the problem of scheduling moldable non-preemptive tasks that arrive in an online fashion. Our objective in this work is to optimize the maximum stretch of such tasks in order to create fair schedules with respect to task sizes. We focus on minimization of instantaneous maximum stretch, which also helps achieving low average stretch values.

Since optimizing the stretch of moldable tasks is a general problem and there is no prior work in the literature, we start by studying the optimization of the maximum stretch of sequential tasks. In Section 2, the sequential task problem is formally presented and lower bounds on the best achievable approximation ratio are provided. In Section 3, two online algorithms `FCFS` and `DASEDF` are presented and their theoretical guarantees are proved.

Furthermore, the use of resource augmentation to improve the reachable approximation ratios is considered. The main theoretical results on the sequential problems are:

- The online optimization of the maximum stretch cannot be approximated strictly better than $\frac{1+\Delta}{2}$ on a single processor (Theorem 2.1) and $\frac{1+\frac{\Delta}{m+1}}{2}$ on m processors (Theorem 2.2), where Δ is the ratio between the processing time of the longest task to that of the shortest task.

- First-Come, First-Served (`FCFS`) is known to be a Δ-approximation algorithm for the single-processor sub-problem. We show that `FCFS` is a $(2\Delta + 1)$-approximation algorithm on m processors (Theorem 3.1).

- We propose an online dual-approximation-based algorithm, called Dual-approximation Algorithm for Stretch using Earliest Deadline First (`DASEDF`), and show each schedule generated by `DASEDF` is a 2-approximation provided that running tasks can not be preempted and no more tasks enters the queue (details in Theorem 3.3). However, its online approximation ratio is at least Δ.

- Resource augmentation [8] is considered. We introduce an algorithm, called `Split`, that takes an $f(\Delta)$-approximation algorithm on m machines and builds an $f(\sqrt[\rho]{\Delta})$-approximation algorithm on ρm machines (Theorem 3.6). It is also shown that no algorithm has an approximation ratio better than $\frac{1+\sqrt[\rho]{\Delta}}{2}$ on ρ machines when the adversary has a single one (Theorem 3.5). Finally, it is shown that using more machines is better than using faster machines (Theorem 3.4 and 3.6).

Through simulations, we demonstrate in Section 4 that `DASEDF` is an order of magnitude better than `FCFS` and the maximum stretch obtained by `FCFS` linearly increases with Δ. We also show that the availability of machines is a key to the problem by showing how reserving machines for tasks with large stretch values reduces the stretch of the tasks.

Section 5 presents moldable task scheduling and a brief discussion of recent studies. In Section 6, we leverage the theoretical and experimental results obtained for the sequential

task model while studying the moldable task model. We design an algorithm, called DBOS, to optimize the stretch of moldable tasks [9]. No strong theoretical guarantees for this algorithm are proposed, however, its efficiency is demonstrated through experiments in Section 7. We show that DBOS leads to stretch values an order of magnitude better than two of the state-of-the-art schedulers in three scenarios: two based on the logs of real clusters and another one that simulates a cluster dedicated for short sequence mapping tasks. Finally, conclusive remarks are provided in Section 8.

2. Sequential Task Scheduling

2.1. Problem Definition

In this section, we describe the problem of online non-preemptive scheduling of sequential tasks on a parallel machine to optimize maximum stretch. The problem is denoted in the three-field notation [10] as $P_m \mid r_i, p_i, online \mid \max s_i$. Consider n tasks to be scheduled on a cluster of m identical processors in an online non-clairvoyant way. Task i is released at time r_i and has a processing time of p_i. The problem is non-clairvoyant: the scheduler does not have any information about the number of tasks to be submitted and is only aware of task i after it is released. Let $p_{max} = \max p_i$ and $p_{min} = \min p_i$ denote the size of the largest and smallest tasks, respectively. Also, let $\Delta = \frac{p_{max}}{p_{min}}$ denote the ratio of the sizes of the largest task to the smallest task.

The scheduler can be invoked at any time. It is also automatically invoked when a task is released or when the execution of a task is completed. For each task i, the scheduler chooses the processor on which the task will be executed as well as the start time $\sigma(i)$ of the execution. A task cannot be scheduled to start before the current time. The tasks can not be preempted: once started, a task must be executed without being delayed, interupted or migrated to an other processor. However, the scheduler can change the start time of the tasks that are not currenlty being executed. A processor can execute only one task at a time.

The completion time of task i can be computed as $C_i = \sigma(i) + p_i$. Then, the stretch of task i is $s_i = \frac{C_i - r_i}{p_i}$. The problem considered in this paper is the optimization of the maximum

stretch, denoted by $\max_i s_i$. As a secondary objective, we also consider the optimization of average stretch, which can be computed as $\frac{\sum_i s_i}{n}$.

2.2. Related Work

The stretch performance index was first introduced in [3] to compensate for the unfairness in database request scheduling, where the focus was mainly on single-machine cases. It was shown that the problem of optimizing the maximum stretch without preemption in an offline setting cannot be approximated within $O(n^{1-\epsilon}), \forall \epsilon > 0$ unless $P = NP$. The importance of the Δ factor was emphasized for preemptive tasks by showing that no $\Omega(\sqrt[3]{\Delta})$ approximation exists. Furthermore, an $O(\sqrt{\Delta})$ approximation algorithm was provided based on the Earliest Deadline First (`EDF`) policy.

The work in [3] inspired several studies in various application contexts with the objective of optimizing stretch with preemption. The Shortest Remaining Processing Time policy to optimize average stretch was studied in [11] and shown to be a 2-approximation on single-machine and a 14-approximation on multi-processor systems. The auhors of [12] reduced a divisible load scheduling problem to a single-machine scheduling problem with preemption and showed that First-Come, First-Served (`FCFS`) is a Δ-approximation algorithm for maximum stretch and a Δ^2-approximation algorithm for average stretch. Using maximum stretch was considered in wireless networks [13] and how to avoid task migration on parallel machines while optimizing stretch was investigated in [14]. The use of resource augmentation techniques (introduced in [8]) and randomization to optimize average flow-time and stretch were studied in [15]. It was shown in [16] that resource augmentation compensates for non-clairvoyance while optimizing stretch.

The literature on non-preemptive scheduling is relatively sparse. The authors are only aware of two works on exotic models. The authors of [17] studied the set scheduling problem without preemption using resource augmentation techniques for transferring files over a network, where the allocated bandwidth for each file can be chosen or changed dynamically measuring the waiting time relative to the size of the transfer. When updating data warehouses in real time, which implies merging updates in an online fashion, the staleness of the

updates (i.e., flow-time) and stretch are considered as the objectives in [18].

2.3. Approximation Lower Bound

In [12], it was shown that the optimization of the maximum stretch on a single-machine with preemption cannot be approximated within $\frac{1}{2}\Delta^{\sqrt{2}-1}$. The proof was given by using the adversary technique [19] which is a commonly used technique in online scheduling problems. Here, we provide a much stronger lower bound on the best reachable approximation ratio for this problem with the additional constraint that the tasks cannot be preempted (Theorem 2.1). Then, we generalize this bound for an arbitrary number of machines (Theorem 2.2). Note that since the proofs are based on the adversary technique, they do not depend on complexity assumptions (such as "if $P \neq NP$").

Theorem 2.1. *All approximation algorithms for the single-processor problem of optimizing the maximum stretch* $1 \mid r_i, p_i, online \mid \max s_i$ *have an approximation ratio of at least* $\frac{1+\Delta}{2}$.

Proof. The proof is based on the adversary technique. Task 1 of size p_1 enters the system at time r_1 and the scheduler schedules it to start at σ_1. Then, the adversary sends task 2 of size $p_2 = p_1/\Delta$ at time $r_2 = \sigma_1$. Since the scheduler cannot preempt task 1 and cannot complete task 2 before $C_2 = \sigma_1 + p_1 + p_1/\Delta$, the stretch of task 2 becomes $s_2 = \frac{\sigma_1 + p_1 + p_1/\Delta - \sigma_1}{p_1/\Delta} = 1 + \Delta$.

It remains to estimate the optimal maximum stretch in an offline scenario. There are three cases to consider. If $r_2 > p_1$, then the optimal maximum stretch is reached by scheduling both tasks immediately after they are released and the maximum stretch is 1. If $p_1 \geq r_2 \geq (1 - 1/\Delta)p_1$, then scheduling task 1 immediately leads to a stretch smaller than 2 for task 2 and a stretch of 1 for task 1. If $r_2 < (1 - 1/\Delta)p_1$, then waiting for task 2 leads to a stretch smaller than 2 for task 1 and a stretch of 1 for task 2. In all cases the optimal maximum stretch is smaller than 2 which leads to the best ratio of $(1 + \Delta)/2$ for any online scheduler. □

Theorem 2.2. *All approximation algorithms for the multi-processor problem of optimizing the maximum stretch* $P_m \mid r_i, p_i, online \mid \max s_i$ *have an approximation ratio of at least* $\frac{1+\frac{\Delta}{m+1}}{2}$.

Proof. The adversary sends batches of m tasks of size Δ every Δ time units until time t at which all machines are scheduled to be busy for at least $\frac{\Delta}{m+1}$ time units. At that time, the adversary sends a task of size 1 in the system and no longer sends any other task. Then, the last task has a stretch of at least $1 + \frac{\Delta}{m+1}$. Using similar arguments as in the previous proof, one can show that the stretch in the optimal schedule is upper bounded by 2 which leads to the ratio of $\frac{1+\frac{\Delta}{m+1}}{2}$.

If no such time t exists, then there is at least one machine idle in each interval of size $\frac{\Delta}{m+1}$. Let I_i represent the length of the i-th time interval in which at least one processor is idle and let W_i represent the length of the interval between idle times i and $i+1$ in which all processors are busy. Let us consider any $m+1$ consecutive idle times. Since there are m processors, at least two of these idle times should occur on the same processor. Moreover, since there is no task of size smaller than Δ, such two idle times should be separated by at least Δ time units. This leads to the fact that the k-th and $(k+m)$-th idle times in a schedule are separated by at least Δ time units for any k. Therefore, $\sum_{i=k+1}^{k+m-1} I_i + \sum_{i=k}^{k+m-1} W_i > \Delta$. Since time t does not exist, $\forall i, W_i < \frac{\Delta}{m+1}$, which leads to $\sum_{i=k+1}^{k+m-1} I_i > \Delta - \frac{m\Delta}{m+1} = \frac{\Delta}{m+1}$. Adding I_k to the left hand side gives $\sum_{i=k}^{k+m-1} I_i > \frac{\Delta}{m+1}$. The ratio of wasted computing power to total available computing power in an interval that contains m idle times can be written as $(\sum_{i=k}^{i=k+m-1} I_i)/(\sum_{i=k}^{i=k+m-1}(I_i + W_i))$. One can show that the lower bound for this ratio is $\frac{1}{m+1}$. Since the overall execution can be partitioned into intervals that contain m idle times, one can conclude that the system is wasting more than $\frac{1}{m+1}$ of its processing power. Therefore, the pending task queue keeps growing and the stretch of the tasks increase linearly with the number of submitted batches. As the number of batches goes to infinity, the maximum stretch of the tasks also goes to infinity. Since the optimal stretch is 1 for this scenario, there is no lower bound for maximum stretch if a time t does not exist. □

Theorem 2.2 is interesting in the sense that the lower bound shown by a similar construction decreases as the number of machines increases, as if the problem becomes easier when the number of processor increases. However, this is a simple generalization for m processors and the bound might be improved.

3. Approximation Algorithms for Sequential Tasks

3.1. First-Come, First-Served: An Approximation Algorithm

`FCFS` has been proven to be a Δ-approximation algorithm for $1 \mid r_i, p_i, pmpt, online \mid \max s_i$ in [12]. Since preemption is not used in `FCFS`, it is also a Δ-approximation algorithm for $1 \mid r_i, p_i, online \mid \max s_i$. In the following theorem we prove a similar result for m machines.

Theorem 3.1. `FCFS` *is a* $\Delta+(1-\frac{1}{m})(\Delta+1)$*-approximation algorithm for* $P_m \mid r_i, p_i, online \mid \max s_i$.

Proof. Let us consider the `FCFS` solution and an optimal solution. The properties of the optimal schedule are denoted with a star, e.g., the completion time of task i in the optimal schedule is denoted by C_i^* whereas it is denoted by C_i in the `FCFS` schedule. Without loss of generality, the tasks are numbered according to their release times. Our goal is to show that for any arbitrary task i, there exists a task j such that $\frac{s_i}{s_j^*} \leq \Delta + (1 - \frac{1}{m})(\Delta + 1)$.

Let t be the first time where a processor is idle before the scheduled start time σ_i of task i. t is set to 0 if no such time exists. The `FCFS` algorithm ensures that the first task to be executed after time t has already been released. Let l denote this task. Then, between r_l and σ_i, the system is completely busy executing tasks k such that $l \leq k < i$, as well as tasks that were released before r_l. Since there are at most $m-1$ tasks released before r_l, $\sigma_i \leq r_l + \frac{\sum_{k=l}^{i-1} p_k}{m} + \frac{(m-1)p_{max}}{m}$. Then,

$$C_i \leq r_l + \frac{\sum_{k=l}^{i-1} p_k}{m} + \frac{(m-1)p_{max}}{m} + p_i. \tag{1}$$

In an optimal schedule, at least one of the tasks between l and i completes after or at

$r_l + \frac{\sum_{k=l}^{i} p_k}{m}$. Now, we estimate the stretch of one such task j in the optimal schedule:

$$
\begin{aligned}
s_j^* = \frac{C_j^* - r_j}{p_j} &\geq \frac{r_l + \frac{\sum_{k=l}^{i} p_k}{m} - r_j}{p_j} && (2)\\
&\geq \frac{C_i - \frac{\sum_{k=l}^{i-1} p_k}{m} - \frac{(m-1)p_{max}}{m} - p_i + \frac{\sum_{k=l}^{i} p_k}{m} - r_j}{p_j} && (3)\\
&\geq \frac{C_i - r_j}{p_j} - \frac{m-1}{m}\frac{p_{max} + p_i}{p_j} && (4)\\
&\geq s_i \frac{p_i}{p_j} - \frac{m-1}{m}\frac{p_{max} + p_i}{p_j}. && (5)
\end{aligned}
$$

In Equation (3), Equation (1) is plugged in for r_l, and r_j in Equation (4) is replaced with r_i since $r_j \leq r_i$. By rearranging Equation (5), the ratio of the stretch can be expressed as:

$$
\begin{aligned}
\frac{s_i}{s_j^*} &\leq \frac{p_j}{p_i} + \frac{m-1}{m}\frac{1}{s_j^*}\frac{p_i + p_{max}}{p_i} && (6)\\
&\leq \Delta + \frac{m-1}{m}(1+\Delta). && (7)
\end{aligned}
$$

The last equation is obtained by remarking that stretch values are at least 1. Equation (7) and the fact that $s_{max}^* \geq s_j^*$ complete the proof. □

The `FCFS` algorithm has a very low complexity. When a task enters the system the scheduler adds it to the list of pending tasks and when the execution of a task completes, the scheduler fetches the task with the smallest release time. Using a double-ended queue both operations can be done in $O(1)$ time. If processors need to be allocated in advance (e.g., to provide data for the tasks), a double-ended queue per processor can be kept up to date, and tasks entering the system can be assigned to the least loaded processor. Using a heap, this information can be obtained in $O(\log m)$ time.

3.2. `DASEDF`*: A Local Approximation Algorithm*

Although `FCFS` has a guaranteed worst case performance, it may result in high stretch values in some situations which could easily be avoided. Consider a single-machine case. If first a large task and then a small task enter an already busy system, `FCFS` schedules the

large task to start before the small task. However, scheduling the small task before the larger one would result in a better maximum stretch. In this section, we propose an algorithm, called `DASEDF` (Dual-approximation Algorithm for Stretch using Earliest Deadline First), to optimize the maximum stretch of the tasks. Even though `DASEDF` does not guarantee a global approximation of the maximum stretch, we show it is a 2-approximation algorithm (or more precisely it has an absolute error [20] less than 1) for (re)scheduling the pending tasks at a given state of the cluster. `DASEDF` is based on dual-approximation [20] and deadline scheduling. Similar algorithms were proposed in [12] for the offline scheduling problem of divisible tasks with preemption and in [3] under the name Dynamic Earliest Deadline First.

`DASEDF` takes a targeted maximum stretch S as a parameter and (re)schedules the pending tasks at a given state of the cluster. Each task i is required to complete before deadline $d_i = Sp_i + r_i$ to match the targeted stretch. Then, it remains to solve the deadline scheduling problem: $P_m \mid p_i, d_j \mid \emptyset$. This problem can be treated with the Earliest Deadline First (`EDF`) algorithm which schedules the tasks as soon as possible in the order of non-decreasing deadlines.

`EDF` is known to schedule all the tasks before their deadlines on a single-machine if such a schedule exists [21]. On m machines, scheduling all the tasks before their deadlines is an NP-Complete problem. However, if a solution that schedules all tasks before their deadlines exists, then `EDF` schedules each task i before $d_i + p_i$. This result is derived from the properties of List Scheduling [22] and is proven below.

Lemma 3.2. *If a schedule that completes each task i before d_i exists, then* `EDF` *creates a schedule where each task i completes before $d_i + (1 - \frac{1}{m})p_i$, where $1 \leq i \leq n$.*

Proof. Without loss of generality, the tasks are numbered in non-decreasing order of deadlines, which is the order in which the tasks are scheduled by `EDF`. In a valid schedule, for each task i, all tasks $i' \leq i$ must be scheduled before d_i. Therefore, at least one task $j \leq i$ must complete after or at $\frac{\sum_{i' \leq i} p_{i'}}{m}$, which implies that $d_j \geq \frac{\sum_{i' \leq i} p_{i'}}{m}$ for a valid schedule. Since $d_i \geq d_j$ due to `EDF` order, $d_i \geq \frac{\sum_{i' \leq i} p_{i'}}{m}$.

In the `EDF` schedule, task i starts as soon as possible after the tasks $i' < i$ are scheduled.

Therefore, it starts before or at $\sigma(i) \leq \frac{\sum_{i' \leq i} p_{i'}}{m}$, hence, $C_i = \sigma(i) + p_i \leq \frac{\sum_{i' \leq i} p_{i'}}{m} + p_i$. Rewriting the last part leads to $C_i \leq \frac{\sum_{i' \leq i} p_{i'}}{m} + (1 - \frac{1}{m})p_i$. Provided the bound on d_i, we have $C_i \leq d_i + (1 - \frac{1}{m})p_i$. □

Note that the proof does not take the tasks currently running on the cluster into account. This can be corrected by adding terms in the d_i and C_i bounds without changing the lemma.

Theorem 3.3. `DASEDF`(S) *returns a solution with maximum stretch less than* $S + 1 - \frac{1}{m}$ *or ensures that no schedule with maximum stretch of* S *exists.*

Proof. The result comes directly from Lemma 3.2. Each task completes before $C_i \leq d_i + (1 - \frac{1}{m})p_i = (S + 1 - \frac{1}{m})p_i + r_i$. Rearranging this inequality gives $\frac{C_i - r_i}{p_i} \leq S + 1 - \frac{1}{m}$. □

Despite the good local property, the approximation ratio of the `DASEDF` algorithm is at least Δ. Indeed, `DASEDF` schedules the instance used in the proof of Theorem 2.2 by executing all the large tasks in parallel and small task is executed after the large ones.

The `EDF` algorithm can be implemented in $O(n \log n)$ time to sort the tasks and $O(n \log m)$ time to schedule them using a heap. The computation of the best S value can be done using a binary search. To check if a value of S is valid, it is sufficient to check the property presented in the proof of Lemma 3.2 ($\forall i, d_i \geq \frac{\sum_{i' \leq i} p_{i'}}{m}$). This can be done in $O(n \log n)$ time to sort the values and in $O(n)$ time to check the property. The binary search must be done between a lower bound and an upper bound. A trivial lower bound LB for the stretch is given by assuming that all the tasks complete immediately. And any valid schedule provides a reasonable upper bound for the optimal value of S (e.g., a schedule given by the `FCFS` algorithm). Notice that scheduling all the tasks sequentially will complete all the tasks before $\sum_i p_i$, which leads to the trivial upper bound $UB = LB + \frac{\sum_i p_i}{\min_i p_i}$. Therefore, using this upper bound, the binary search performs at most $\log(\frac{\sum_i p_i}{\min_i p_i})$ steps which is polynomial in the size of the problem.

3.3. Resource Augmentation

Resource augmentation technique was introduced in [8], and it is used to analyze an online scheduling algorithm when additional computing resources are available. The idea is to

evaluate the amount of additional resources required to overcome the limitations of an online algorithm. In more technical terms, the optimal solution is determined by the amount of available resources (such as number of machines, speed of the machines, network bandwidth, memory, etc.), while the algorithm is applied on an instance that has more resources available. Yet, the performance of the algorithm is expressed relative to the optimal un-augmented solution. The technique was first applied on a single-machine problem [23], then extended to multiple-machine cases to optimize flow-time and to satisfy deadline requirements [15].

Resource augmentation can be applied either by considering ρ times faster machines without changing the number of machines, or by considering ρ times more machines of the same speed. In this section, we evaluate the impact of resource augmentation on reducing the Δ factor of the problem. It is assumed that the value of ρ is known to the adversary. Thus, the adversary can adapt its strategy based on ρ. In particular, the adversary can choose Δ while having the information on the value of ρ.

3.3.1. Faster Machines

We first consider resource augmentation using ρ times faster machines. In the following theorem we derive a bound on the best achievable approximation ratio for the single-machine case by adapting the adversary scenario.

Theorem 3.4. *All approximation algorithms using a ρ times faster machine for the problem $1 \mid r_i, p_i, online \mid \max s_i$ have an approximation ratio of at least $\frac{1+\Delta}{2\rho}$.*

Proof. The proof is very similar to the proof of Theorem 2.1. Task 1 enters the system at time r_1 and has a processing time of p_1. The scheduler executes the task at time σ_1. The adversary submits task 2 of size $p_2 = p_1/\Delta$ at $r_2 = \sigma_1$. Since the scheduler cannot preempt task 1, task 2 cannot finish before $C_2 = \sigma_1 + p_1/\rho + p_1/(\rho\Delta)$, leading to $s_2 = \frac{\sigma_1+p_1/\rho+p_1/(\rho\Delta)-\sigma_1}{p_1/\Delta} = \frac{1+\Delta}{\rho}$.

The optimal stretch for a single-machine case at normal speed is at most 2 (see the the proof of Theorem 2.1 for details). Therefore the best achievable approximation ratio is $(1 + \Delta)/2\rho$. □

The theorem shows that having a ρ times faster machine does not help breaking the Δ

factor in the approximation ratio. Therefore, we do not investigate this case any further.

3.3.2. More Machines

We first consider resource augmentation using ρ times more machines to improve the maximum stretch. The following theorem is an adaptation of the single-machine adversary technique using ρ machines.

Theorem 3.5. *All approximation algorithms using ρ machines for the problem $1 \mid r_i, p_i, online \mid \max s_i$ have an approximation ratio of at least $\frac{\sqrt[\rho]{\Delta}}{\rho+2}$.*

Proof. The adversary generates $\rho+1$ tasks of sizes $\Delta, \sqrt[\rho]{\Delta}^{\rho-1}, \ldots, \sqrt[\rho]{\Delta}^{\rho-i}, \ldots, \sqrt[\rho]{\Delta}, 1$ where $\Delta \gg 2^\rho$. They are submitted in decreasing task size order and each task is released as soon as the execution of the previous one is scheduled.

By contradiction, we now prove that a non-clairvoyant scheduler does not get a maximum stretch lower than $\sqrt[\rho]{\Delta}$. Assume that the stretch of all tasks is less than $\sqrt[\rho]{\Delta}$. Then, the waiting time of task $i \geq 1$ is $C_i - r_i = s_i p_i < \sqrt[\rho]{\Delta}\sqrt[\rho]{\Delta}^{\rho-i} = p_{i-1}$. Since task i is released when task $i-1$ begins its execution and since it should wait less than p_{i-1} time units, task i needs to be executed concurrently with task $i-1$. Recursively, task i needs to be executed concurrently with all tasks $i' < i$. However, there are $\rho+1$ tasks but only ρ processors. Therefore, at least one of the tasks has to be postponed, which results in a maximum stretch of at least $\sqrt[\rho]{\Delta}$.

We now show that the optimal solution has a stretch lower than $\rho+2$ by considering a valid solution of maximum stretch lower than $\rho+2$. First, note that the sum of processing times of k smallest tasks is smaller than the processing time of the $(k+1)$-th smallest task (since $\Delta > 2^\rho$). That is, $\sum_{i=0}^{k} p_i = \sum_{i=0}^{k} \sqrt[\rho]{\Delta}^i = \frac{\sqrt[\rho]{\Delta}^{k+1}-1}{\sqrt[\rho]{\Delta}-1} \leq \sqrt[\rho]{\Delta}^{k+1} = p_{k+1}$. Now, consider an offline algorithm that greedily schedules the tasks as soon as possible in increasing processing time order. Each scheduled task introduces at most one idle time in the schedule. Each task waits at most for the duration of ρ idle times since there are $\rho+1$ tasks and each idle time is smaller than its processing time due the as-soon-as-possible rule. It also waits during the execution of smaller tasks (due to the ordering): $C_i - r_i \leq \rho p_i + \sum_{i'=0}^{i} p_{i'} + p_i$. This leads to the following bound on the stretch: $s_i = \frac{C_i - r_i}{p_i} \leq \rho + 2$. □

The theorem exhibits a weaker lower bound on the approximability of the machine augmented problem and we introduce the `Split` algorithm to use effectively more machines. `Split` can use any algorithm `ALGO` for the online problem. `Split` considers the ρm machines as ρ independent clusters. It is assumed that p_{min} and p_{max} are known. When a task i enters the system, it is scheduled on the $(j+1)$-th cluster such that $p_{min}\sqrt[\rho]{\Delta}^{j} \leq p_i \leq p_{min}\sqrt[\rho]{\Delta}^{j+1}$.

Algorithm 1 `Split`

Require: p_{min} and p_{max}: the range of the processing times and $\Delta = \frac{p_{max}}{p_{min}}$, m: the number of machines, $\rho \geq 2$: the augmentation factor, `ALGO`: an online scheduling policy for m machines.

1: Partition the ρm machines into ρ groups: $(0, m-1), (m, 2m-1), \ldots, ((\rho-1)m, \rho m - 1)$
2: Assign online policy $\texttt{ALGO}_j$ to $(jm, (j+1)m-1)$ for $0 \leq j \leq \rho$
3: **while** task i is released or completed **do**
4: Let $j \in \mathbb{N}$ be such that $p_{min}\sqrt[\rho]{\Delta}^{j} \leq p_i \leq p_{min}\sqrt[\rho]{\Delta}^{j+1}$
5: Transfer event to $\texttt{ALGO}_j$

Theorem 3.6. *If* `ALGO` *is an* $f(\Delta)$*-approximation algorithm on* m *machines,* `Split` *is an* $f(\sqrt[\rho]{\Delta})$*-approximation algorithm on* ρm *machines using* `ALGO`*.*

Proof. Let I represent the instance of the original problem. Let i be a task that is scheduled by `Split` on the $(j+1)$-th cluster. Index j is such that $p_{min}\sqrt[\rho]{\Delta}^{j} \leq p_i \leq p_{min}\sqrt[\rho]{\Delta}^{j+1}$.

What happens on cluster $j+1$ can be seen as another instance of the scheduling problem. Let I' represent the instance of the online scheduling problem on m processors with the tasks allocated to the $(j+1)$-th cluster. By construction of I', the ratio between the largest task and the smallest task is $\sqrt[\rho]{\Delta}$ and since `ALGO` is an $f(\Delta)$-approximation algorithm, we have $s_i \leq f(\sqrt[\rho]{\Delta})s^*_{max}(I')$. The optimal stretch of I' is smaller than the optimal stretch of the original instance I since I' has less tasks and the same number of processors. Therefore, $s^*_{max}(I') \leq s^*_{max}(I)$. Plugging both expressions together leads to, $s_i \leq f(\sqrt[\rho]{\Delta})s^*_{max}(I)$. □

Using `FCFS`, `Split` results in a $(2\sqrt[\rho]{\Delta}+1)$-approximation algorithm of the ρ resource augmented problem. This indicates that availability of machines is the key to overcome the Δ barrier.

4. Experimental Evaluation of Methods for Sequential Tasks

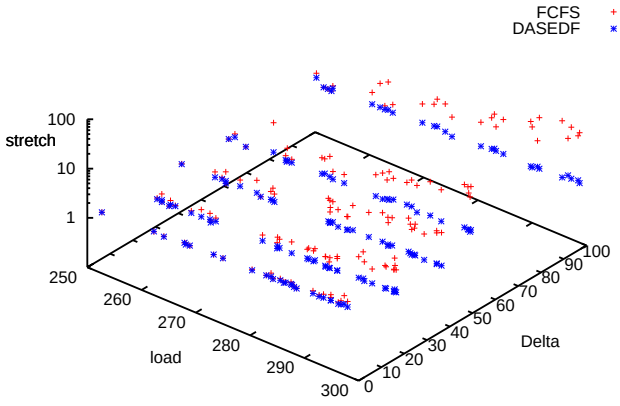
In this section, we report on our experiments using `FCFS` and `DASEDF` under various online scheduling scenarios to evaluate more practical aspects of the algorithms. The experiments are designed to assess the effects of system load, Δ, and machine availability. Load and Δ are the two main parameters that define a scheduling instance, where load is defined as the ratio of aggregate processing times of all tasks to the time elapsed between the release of the first and the last tasks ($\frac{\sum_i p_i}{\max_i r_i - \min_i r_i}$). In other words, for a load of l, total computing power of l processors would be used to execute the tasks over the time for which the activity on the cluster is simulated.

Regarding machine availability, we designed a machine reservation scheme that splits the cluster in two parts: the main part with $m - X$ machines and the auxiliary part with X machines. Then, we tested whether having a less busy auxilary part which is used only when a new task results in a schedule with large maximum stretch in the main part helps improving the overall maximum stretch. In this scheme, when a task is released, a new schedule is computed for the main part including the new task. If the maximum stretch in the resulting schedule is less than a threshold T, then the schedule is confirmed and no further action is taken. Otherwise, a new schedule is also computed for the auxilary part, including the new task, to see if it leads to a smaller maximum stretch. Then the task is assigned to the part that results in a schedule with smaller maximum stretch and the changes in the other part are reverted.

In the remainder of this section, experimental setup and results are presented.

4.1. Setup

We generate a variety of problem instances, each comprises of $20,000$ tasks to be executed on 300 processors. The sizes of the tasks are randomly chosen from a uniform distribution over the interval $[a; b]$; and the release times of the tasks are generated such that the inter-arrival time follows an exponential distribution with parameter λ. The parameters λ, a and b are chosen to obtain Δ values of 5, 10, 15, 20, 40, 60, 80 and 100 and load values between 220 and 310. There are 8 different values for the (a, b) couple and between 6 and



(a)

Δ	Nb inst.	FCFS	DASEDF
5	50	1.80(0.94)	1.42(0.27)
10	76	4.45(4.38)	1.70(0.65)
15	64	2.95(1.76)	1.40(0.17)
20	60	4.45(3.66)	1.46(0.27)
40	61	11.20(13.90)	1.61(0.52)
60	44	15.10(17.60)	1.60(0.41)
80	60	20.90(25.40)	1.69(0.45)
100	49	28.90(37.30)	1.77(0.53)

(b)

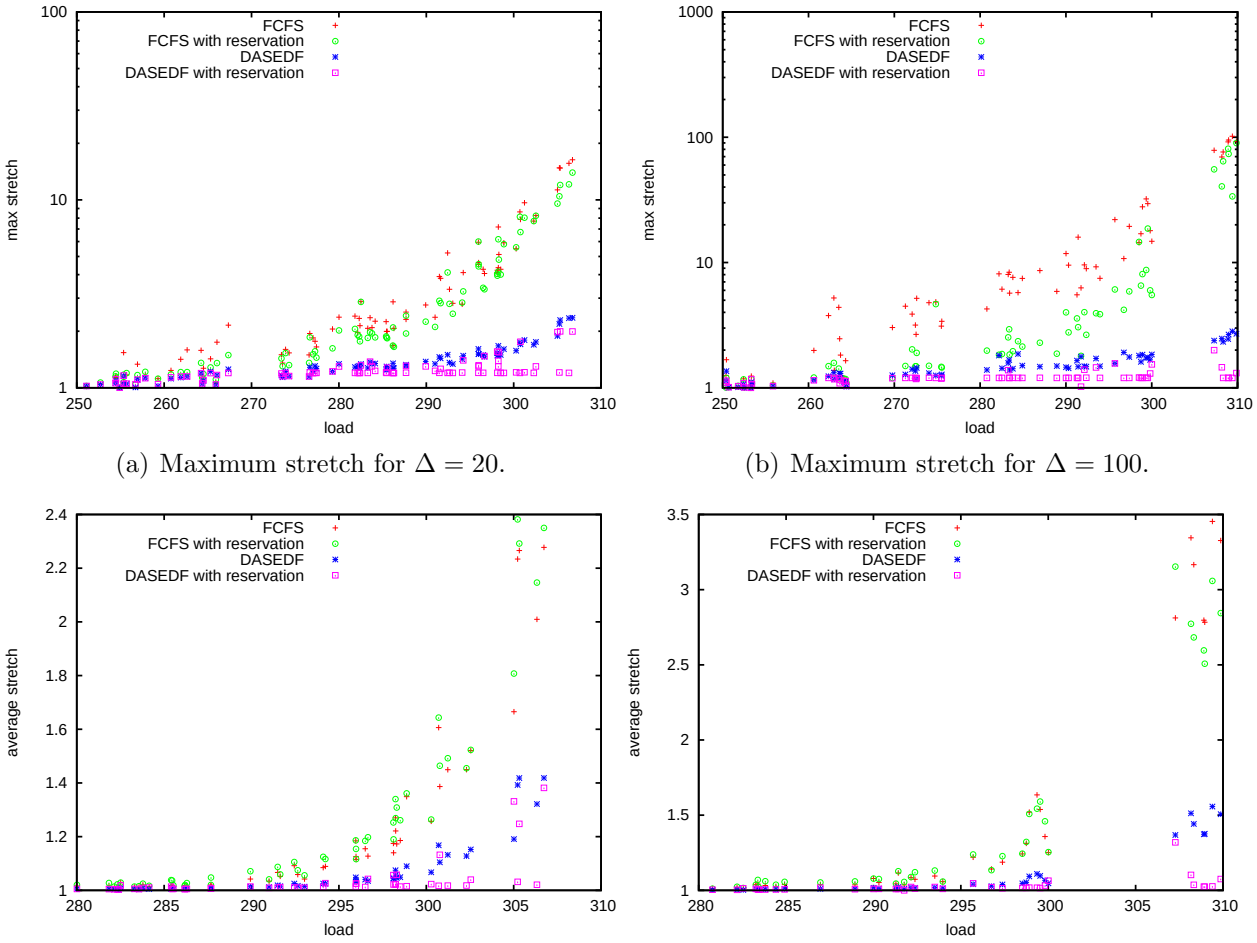
Figure 1: Maximum stretch results for FCFS and DASEDF: (a) as a function of load and Δ, (b) mean and standard deviation (between parenthesis) of maximum stretch for various Δ values, over instances with load > 270 (their number is reported in the second column) without reservation.

10 different values of λ for each (a, b) couple. Each set of parameter is used to generate multiple instances, leading to 670 instances in total. All the charts present actual results: no aggregation of multiple instances have been performed. In other words, one point is the result given by one algorithm on one instance.

For the machine reservation scheme, the tested values of X were 1, 2, 5, 10, 15, 20 and 30. The tested values of T for FCFS are 1.2, 1.5, 1.8, 2, 2.5, 3, 4, 6, 8 and 10, and those for DASEDF are 1.2, 1.3, 1.4, 1.5, 1.6, 1.8, 2, 2.5 and 3. Because our goal is to assess whether the reservation scheme can lead to significant improvement or not, we only report results for the best combination of X and T values that give the lowest maximum stretch for each instance.

4.2. Results

Figure 1(a) presents the maximum stretch (in log scale) obtained by FCFS and DASEDF on the tested values of Δ and the load (some Δ values and load values greater than 300 are omitted for clarity). The results indicate that maximum stretch increases with an increase in any of those parameters. However, the difference between the algorithms is negligible for the instances with load less than 270. Mean and standard deviation of the maximum stretch over the instances with load greater than 270 are given in Table 1(b) as a function of Δ. Both of these quantities grow almost linearly with Δ when using FCFS, whereas they



(a) Maximum stretch for $\Delta = 20$.

(b) Maximum stretch for $\Delta = 100$.

(c) Average stretch for $\Delta = 20$.

(d) Average stretch for $\Delta = 100$.

Figure 2: Maximum and average stretch as a function of load for $\Delta = 20$ and $\Delta = 100$.

were more stable and increased slightly for DASEDF. These results imply that, although the approximation ratios for both FCFS and DASEDF are shown to be linear in Δ, contrary to FCFS, such worst case scenario occurs rarely for DASEDF. Furthermore, the largest maximum stretch over all DASEDF solutions is 2.5, whereas for FCFS there are many instances with maximum stretch greater than 10 especially for larger Δ values.

In Figure 2, the maximum and average stretches obtained when using FCFS, DASEDF, and their variants that employ the machine reservation scheme are presented for varying load values. The results in Figures 2(a) and 2(c) correspond to the test instances with $\Delta = 20$, and those in Figures 2(b) and 2(d) correspond to the test instances with $\Delta = 100$.

The results show that DASEDF significantly outperfoms FCFS in both maximum and average stretch in all test cases.

Furthermore, it is observed that the machine reservation scheme helps reducing the maximum stretch of both algorithms, where the improvement for FCFS is more pronounced for large Δ values. Indeed, for high load values (greater than 270), the reservation scheme improved the maximum stretch of FCFS in 55 cases over 60 instances for $\Delta = 20$, and in 48 cases over 49 instances for $\Delta = 100$.

On the other hand, for FCFS, the cost of improvement in maximum stretch is a potential degradation in average stretch when using the machine reservation scheme. For high load values (greater than 290), the reservation scheme degrades the average stretch of FCFS in all 31 instances for $\Delta = 20$. The situation is not so clear cut when $\Delta = 20$ where the reservation scheme degrades the average stretch in 16 cases over 29 instances. An observation of Figure 2(d) shows that the difference in average stretch can be large in both sense. This behavior is caused by overloading of the main part of the cluster, which manifests itself by an exponential increase in average stretch values for increasing load. Another partial reason is that the values of X and T are chosen to optimize the maximum stretch of an instance, without considering the average stretch.

When using DASEDF, however, the machine reservation scheme helps also reduce the average stretch significantly, for load greater than 290, in 25 cases over 31 instances for $\Delta = 20$ and in 25 cases over 29 instances when $\Delta = 100$). This can be explained by the fact that every time the scheduler is invoked DASEDF attempts to minimize the *instantaneous* maximum stretch rather than only targeting to keep the current maximum stretch below the global maximum stretch acheived so far. As a result, processor utilization and average stretch are also improved in addition to maximum stretch.

We have also measured the aggregate runtime of the algorithms for scheduling all the tasks in each scheduling instance. On the average, FCFS and DASEDF take 15 and 40 seconds, respectively, to schedule all of the $20,000$ tasks in an instance. The longest time required to schedule an instance is 107 seconds for FCFS and 553 seconds for DASEDF. This means

that, even in the worst case, the runtime of `DASEDF` was 26 milliseconds per task submission. Therefore, the runtime of the algorithms are unlikely to become a bottleneck in a production environment.

5. Moldable Task Scheduling

In this section we discuss details about online scheduling of moldable parallel tasks. We use the same online setting and notation introduced in Section 2.1 with a few adaptations for parallel tasks. The scheduling problem considered here is to decide the number of processors π_i to be allocated for each task i and the task's start time σ_i on the system. We use the notation $p_{i,j}$ to represent the execution time of task i on j processors. Therefore, the completion time C_i of task i becomes $C_i = \sigma_i + p_{i,\pi_i}$. The definition of stretch is also slightly modified. We define the stretch of a parallel task i as $s_i = \frac{C_i - r_i}{p_{i,1}}$.

To the best of the authors' knowledge, stretch has never been used, nor defined in the context of online moldable scheduling without preemption before. The only related work in that respect was given in [11], where stretch was used for online scheduling of rigid parallel tasks with preemption.

The adversary technique presented in Theorem 2.2 can easily be extended to moldable tasks by using tasks whose processing time on j processors is the same as the processing time on one processor. Note that this technique is only of theoretical consideration since it is unlikely that sequential tasks would be scheduled using a moldable task scheduler. The moldable task model allows one large task to be scheduled on the whole cluster. Such an allocation is a worst case since it is likely to induce a very large stretch on all the upcoming tasks.

In the task scheduling literature, objective functions similar to stretch have also been used. A commonly used objective function is the slowdown of a rigid task, which is the time the task spends in the system divided by its processing time. Since the task is rigid, the processing time is the same as the actual execution time. As a result, the slowdown is always greater than 1. Stretch can be considered as an extension of slowdown for the moldable task model.

A variant of the slowdown objective function is Bounded slowdown (BSLD) [5] which is used to avoid over-emphasizing the significance of small tasks. In BSLD, the processing time of the tasks are assumed to be greater than a given constant. Since this may result in some tasks to have a slowdown less than 1, the slowdown values between 0 and 1 are rounded up to 1. Due to the rounding, BSLD is not appropriate for the moldable task model, as the stretch or slowdown values of interest for this model can be less than 1.

Another closely related objective function is the Xfactor [24], which is defined as $\frac{t-r_i+p_{i,1}}{p_{i,1}}$, where t is the current time. Xfactor is always greater than 1 and it does not take the number of processors used to execute a task into account.

In the following subsections, we describe principles and algorithms to schedule rigid or moldable tasks. The *backfilling* principle aims at scheduling rigid tasks as soon as possible to optimize the system utilization and the flow time of a particular task. *Batch scheduling policies* are commonly used to optimize the average flow-time. The *fair-share scheduling algorithm* optimizes average flow-time but can include some concern of the Xfactor. The *iterative moldable scheduling algorithm* explicitely optimizes average flow-time. Finally, *SCOJO-P* mainly optimizes the utilization of the system and considers the flow-time of particular tasks as a secondary objective. Notice that none of these principles and algorithms explicitely optimizes stretch.

5.1. Backfilling

Before discussing related work in the rest of this section, we give a brief overview of *backfilling* strategies used by many scheduling algorithms to improve cluster utilization. In most algorithms, tasks are scheduled as soon as possible in the order of their arrival times. This results in holes in the schedule which can be utilized later by using a conservative or an aggressive backfilling strategy. In conservative backfilling, a task is scheduled in the first hole that can accommodate the task. If no such hole exists, the task is scheduled at the end of the schedule. In aggressive backfilling, a task is scheduled in the first hole that has enough number of available processors. If this creates a conflict, then the task in conflict with the largest start time is rescheduled. This approach provides better utilization of the

cluster by reducing the number and size of the holes in the schedule. However, it tends to reschedule large tasks several times, causing longer delays for them. Further details on backfilling strategies can be found in [25].

5.2. Batch scheduling policies

In moldable task scheduling, the most studied objective functions are based on aggregation of completion time (makespan) of the tasks, e.g., minimization of maximum completion time and minimization of average completion time. A common technique to optimize completion time is to use dual-approximation [26, 27]. This technique consists of choosing a target value for the objective and then deciding the most efficient number of processors a task should use to finish before the targeted completion time. Such number of processors is commonly called the canonical number of processors. A major disadvantage of using completion time in the objective function is the requirement of a time origin, which does not suit well to online scheduling problems.

These offline algorithms are transformed into online algorithms by scheduling tasks in batches. The scheduler is invoked only when the batch of tasks scheduled in the last invocation complete their execution. Then, all pending tasks (i.e., tasks released after the last invocation) are scheduled to be executed in the next batch. These techniques are commonly used to optimize the sum of completion time or the average flow-time [28].

5.3. The fair-share scheduling algorithm

The fair-share scheduling algorithm has been proposed in [29] and refined in [30] to optimize the average flow-time. Note that algorithms that use a flow-time-based objective function do not need a time origin as opposed to batch scheduling policies. The basic principle of the original fair-share algorithm is to greedily schedule tasks one by one to minimize their completion time using aggressive backfilling. This approach leads to executing tasks in parallel using all processors in the system and therefore results in low efficiency. To avoid this scenario, the fair-share algorithm limits the maximum number of processors that can be allocated to each task. This limit is called the fair-share limit, and finding a good value for the

fair-share limit is the motivation behind the mentioned studies. The fair-share limit of a task i was initially set to the ratio of work associated with the task to the total work associated with all tasks pending in the system. Using this limit is stated to be fair since it allocates more processors to larger tasks while limiting the maximum allocation by the weight of the tasks. It was shown that using a fair-share limit of $\frac{\sqrt{p_{i,1}}}{\sum_k \sqrt{p_{k,1}}}$ leads to best results. However, this value was later reported to be too restrictive and so was multiplied by an overbooking factor in [30] to allow the scheduler to consider a larger number of possibilities.

The fair-share algorithm induces starvation due to aggressive backfilling which can delay all tasks but the first one. Therefore, tasks are partitioned in multiple queues based on the their sizes. Starvation is reduced by ensuring the first task of each queue is never delayed. To further reduce starvation, the Xfactors of the tasks are also considered. If the Xfactor of a task exceeds a threshold, it is not allowed to be rescheduled by backfilling.

5.4. Iterative moldable scheduling algorithm

The fair-share algorithm provides fairly good performance but requires tuning many parameters. In [30], Sabin *et al.* proposed a parameter-free iterative scheduling technique which is reported to outperform the fair-share algorithm and its variants. The fundamental idea in the algorithm is to make all tasks rigid by deciding the number of processors to be allocated for each task. Then, the tasks are scheduled using conservative backfilling. The order in which the task are considered for backfilling is not given in [30]. In the following we assume that FCFS order is used.

To determine the number of processors to allocate for each task, the algorithm starts by allocating one processor to each task and computes the corresponding schedule. Then, the task that would have the most reduction in its processing time by using an extra processor is determined. Subsequently, an additional processor is assigned to that task and a new schedule is computed. If the new schedule has a better average flow-time, then the extra processor allocation is confirmed and the process is repeated iteratively. Otherwise, the algorithm rolls back to the previous allocation state and never tries to assign an additional processor to that task again.

The algorithm implicitly assumes that the processing time of a task strictly decreases with the number of processors. However, this assumption may not hold in practice. For example, it is fairly common that parallel algorithms require a number of processors which is a power of two, which induces steps in the speedup function. Moreover, in the short sequence mapping problem that motivates our study, if the number of processors m is prime, it is likely that using fewer processors than m results in a better runtime.

Improvements to the algorithm: Existence of steps in the speedup function results in early termination of the iterative scheme in [30]. To remedy this situation, we propose the following modification. If task i is allocated x processors, instead of considering its execution on $x+1$ processors, we consider its execution on $x+k$ processors ($k \geq 1$) such that $\frac{p_{i,x}-p_{i,x+k}}{k}$ is maximal. If the speedup function is convex, this modification behaves the same as the original algorithm. If the speedup function is not convex, the modification allows to skip the allocation sizes that would lead to low efficiency. Throughout the paper, we refer to this variant of the algorithm as the *improved iterative* algorithm.

5.5. SCOJO-P

SCOJO-P scheduler proposed in [31] aims at optimization of cluster utilization, however, both flow-time and BSLD are considered in the experimental evaluation. The BSLD of a task is defined relative to the processing time on *optimal* number of processors, for which the runtime to efficiency ratio is minimized.

Each task is initially allocated an *optimal* number of processors. Then, the processor allocation is scaled so that around 95% of the processors are kept busy during the execution of the task. To estimate the percentage of processors that will be busy during the span of the task's execution, SCOJO-P maintains a model of the workload of the system using recent history. This model considers the tasks that are predicted to arrive in the system, in addition to those that are already in the system. The task is then scheduled to start immediately, if possible. Otherwise, it is considered for scheduling using fewer processors if it improves its flow-time. Conservative backfilling is also considered in SCOJO-P to improve the efficiency of the system.

6. Deadline Based Online Scheduling

In this section, we present an algorithm, called *Deadline Based Online Scheduling (DBOS)*, to solve the problem of optimizing the maximum stretch for moldable tasks [9]. `DBOS` is based on the lessons learned from the study of the following scheduling of sequential tasks:

- the problem of optimizing the stretch of released tasks can be reduced to the problem of meeting deadlines;
- machine availability helps dealing with upcoming tasks.

Algorithm 2 DBOS

```
 1: procedure DBOS(INPUT: ρ, OUTPUT: π*, σ*)
 2:     (π*, σ*) ← DASEDF(MoldableEDF).
 3:     Let S be the maximum stretch of (π*, σ*)
 4:     (π^ρ, σ^ρ) ← MoldableEDF(ρS)        ▷ Relax S by a factor of ρ if it is feasible
 5:     if Feasible (π^ρ, σ^ρ) then
 6:         (π*, σ*) ← (π^ρ, σ^ρ)
 7:     return (π*, σ*)
 8:
 9: procedure MoldableEDF(S)
10:     for all i ≤ n do                    ▷ Compute a deadline for each task
11:         D_i ← r_i + p_{i,1} S
12:     Construct initial processor allocation using information about running tasks
13:     for all task i in non-decreasing D_i order do
14:         for all j from 1 to m do
15:             x ← earliest time that j processors are available for p_{i,j} units of time
16:             if x + p_{i,j} ≤ D_i then
17:                 π_i ← j; σ_i ← x
18:                 Exit inner for loop
19:     return (π, σ)
```

The outline of the `DBOS` algorithm is presented in Algorithm 2 (lines 1–7). `DBOS` estimates the best feasible maximum stretch by calling the `DASEDF` algorithm discussed in Section 3.2. However, when solving the deadline problem in `DASEDF`, the `MoldableEDF` procedure is used instead of its sequential counterpart to address the moldable nature of the problem (line 2). The maximum stretch obtained by `DASEDF` is then relaxed by a user provided factor of ρ (called *the online factor*) to improve machine efficiency and availability for upcoming tasks.

Finally, `MoldableEDF` is called once again to compute the relaxed schedule, and the new schedule is saved if it is feasible (lines 4–6).

The details of the `MoldableEDF` procedure is given in Algorithm 2 (lines 9–19). Given a maximum stretch value of S, `MoldableEDF` starts by computing a deadline $D_i = r_i + p_{i,1}S$ for each task i (lines 10–11), which reduces the problem to scheduling the tasks before their deadlines. Then, the tasks are scheduled greedily in non-decreasing order of their deadlines (line 13). For each task i, the smallest number of processors j that allows the task to finish before its deadline D_i without moving any previously scheduled task is determined. Finally, task i is scheduled to start as soon as possible on j processors. If it is not possible to schedule task i before D_i, the constructed schedule is labeled as infeasible. Remark that the core of the deadline scheduling algorithm from line 12 to line 18 is generic. It could be used for a classical scheduling problem of moldable tasks with deadlines.

`MoldableEDF` is a greedy algorithm and can fail to find a feasible solution. However, `MoldableEDF` is based on two principles that make it efficient. First, the tasks are considered in non-decreasing order of deadlines which has been shown to result in good schedules while scheduling sequential tasks (see Section 3.2). Second, the algorithm allocates the minimum number of processors that ensures a task matches its deadline. This principle is similar to the canonical number of processors used in makespan optimization and has two advantages. First, the algorithm is oblivious to the shape of the speedup function. And second, it maximizes the processor availability for the other tasks in the system which helps keeping the system efficiency high.

Multiplying the obtained "best" feasible value of S by ρ in Algorithm 2 increases the efficiency of the system further and leaves potentially more processors for the tasks that will arrive in the future. Furthermore, this helps improving the performance in a scenario where a large task enters the system followed by small tasks (similar to the worst case scenario used in the adversary technique). The online factor ρ is the key to the online aspect of the `DBOS` algorithm and shares the same goal as the reservation scheme discussed in Section 4. Since the online factor is a single parameter, it is expected to be easier to tune than the

reservation scheme.

7. Experiments

In this section, we report on the simulation results of the `DBOS` algorithm on a 512-processor cluster using three workload scenarios that reflect variety in task execution times and system load. The results of `DBOS` are presented in comparison to the `Iterative` algorithm of Sabin *et al.* [30] and `SCOJO-P` [31], which were described in Sections 5.4 and 5.5, respectively. The simulator is written in C++. Both `DBOS` and the `Iterative` algorithms are natively written in this simulator and a JAVA implementation of `SCOJO-P` (provided by Dr. Sodan) is interfaced to the simulator.

In the first two workload scenarios, we use two real log files (SDSC Par 96 and OSC Cluster in [32]) of parallel jobs submitted to the San Diego Supercomputing Center (SDSC) and to the Ohio Supercomputing Center (OSC) to assess the performance of the algorithms on well-known data sets. The log files contain information about each task's arrival time, runtime on the system and the number of processors used for its execution. We considere the first $8,000$ tasks in the log files, and similar to the procedure in [30], we use the Downey model [33] to estimate the scalability of the tasks. The Downey model requires two parameters for each task: maximum parallelism and variance of parallelism of the task. The value of the maximum parallelism is randomly selected between p and 512, where p is the recorded number of processors used to execute the task in the log file. The value of the variance of parallelism is randomly selected between 0 and 2 which is a realistic range for this parameter [33]. Since the Downey model is stochastic, 20 different instances are generated for each log file. The average loads (as defined in Section 4) for OSC and SDSC logs are 12.4 and 272.4, respectively. The ratio of the sequential processing time of the largest task to that of the smallest task is 6.2×10^7 for OSC logs and 1.6×10^8 for SDSC logs.

The third workload scenario is designed to simulate tasks from the short sequence mapping application that motivated this study. Full description of the task properties for this application can be found in [9]. Here, we give a summary of the main properties. A task

in this scenario represents a mapping operation of short sequences generated by a next-generation sequencing machine to a reference genome. The processing time of such a task depends on the size of the reference genome and the number of short sequences to be mapped. The processing times of different tasks can vary greatly. For instance, a targeted sequence analysis involves mapping a few million short sequences to a genome segment of a few hundred thousand bases and can be carried out in a couple of minutes. On the other hand, a whole-genome resequencing application requires mapping hundreds of millions of short sequences to the human genome and may take a few days.

The parallelization of a short sequence mapping task is carried out by expressing the computations as a three dimensional space. The work in each dimension is partitioned into x, y and z parts to achieve parallel processing on xyz processors. Given the size of the genome, the number of short sequences to be mapped, and the x, y and z parameters, the parallel runtime of a short sequence mapping task can be predicted accurately. The runtime on p processors is estimated by selecting x, y and z with $xyz \leq p$, such that the runtime is minimized. Notice that, for a given p, since the best way of parallelizing a task may require less than p processors there are steps in the speedup function which may impact the performance of the scheduling algorithms. For a complete explanation of the parallelization techniques for short sequence mapping tasks, please refer to [6].

Each workload in this scenario consists of 8,000 parallel short sequence mapping tasks. The genome size and the number of short sequences associated with a task are selected by following the same procedure as in [9]. The sequential processing time of the generated tasks vary between 30 seconds and 2.5 days inducing a ratio of 5784 between the largest and the smallest processing times. Each task arrives at the cluster with an inter-arrival time chosen from a exponential distribution of parameter λ_i. We vary λ_i to obtain seven different load conditions. For each workload condition 20 instances were generated. Since the instances are generated stochastically, the load of each instance is different from the others. Therefore, we consider a range of load values around a targeted load value (and tuned the λ_i parameter to reach such load). The seven load cases in the experiments correspond to the following

ranges of load values: 153-165, 203-220, 256-280, 332-359, 386-410, 410-450 and 470-509.

7.1. Results on instances from real cluster log files

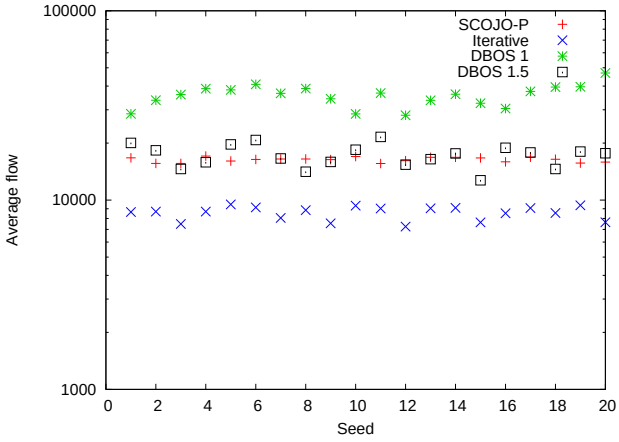
The results of the 20 simulations using the Downey model on the SDSC log file are presented in Figure 3 for `SCOJO-P`, `Iterative` and two versions of the `DBOS` algorithm. In the version represented by `DBOS 1`, the online factor is deactivated, whereas in the version represented by `DBOS 1.5`, the online factor is set to 1.5. In the first four charts, the y-axis is in log scale and different instances are represented by different seed values on the x-axis.

The results in Figure 3(a) show that the `Iterative` scheduler leads to an average flow-time of around 9000 seconds (2.5 hours) whereas `SCOJO-P` performs approximately twice worse. Without the online factor, tasks scheduled by `DBOS` get a flow-time 4 times worse than `Iterative` on the average. With an online factor of 1.5, `DBOS` matches the performance of `SCOJO-P`.

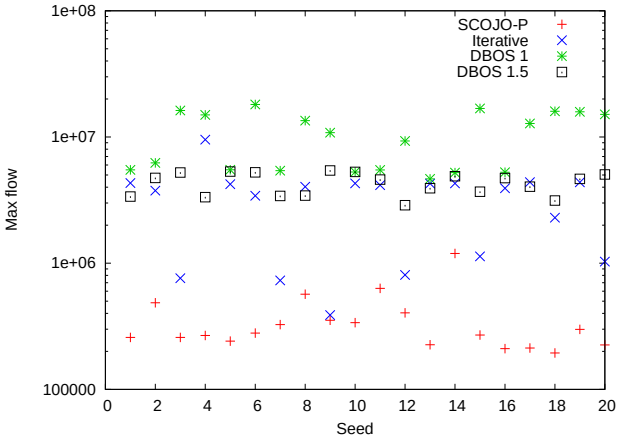
In terms of maximum flow-time (Figure 3(b)), the results are more erratic. Using `SCOJO-P` leads to the best maximum flow-time in all of the 20 tested cases (around half a day). The iterative scheduler gets a maximum flow-time 10 times worse than `SCOJO-P` and `DBOS` performs even worse without the online factor. When the online factor is set to 1.5, however, `DBOS` matches the performance of the `Iterative` algorithm in most cases.

The results on average stretch (Figure 3(c)) indicate that `DBOS` with an online factor of 1.5 leads to the best average stretch in majority of the test cases. In the remaining cases, `DBOS` without the online factor results in the smallest average stretch. The average stretch of both `DBOS` versions have a large variance over different workload instances, but it is usually between 1 and 10. The `Iterative` algorithm results in an order of magnitude worse average stretch than `DBOS` (around 50). And `SCOJO-P` leads to an average stretch two orders of magnitude worse than `DBOS` (around 500).

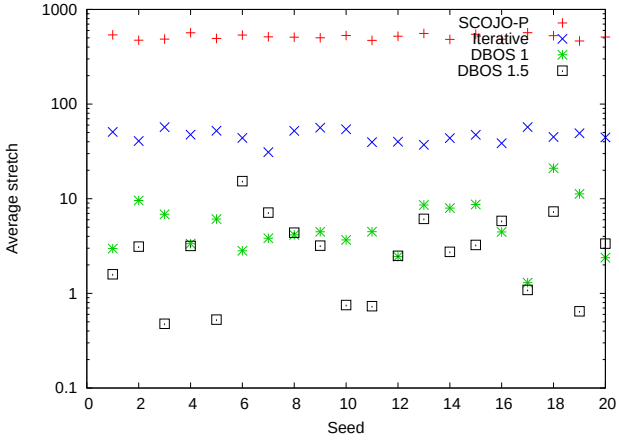
The algorithms are ranked similarly according to maximum stretch (Figure 3(c)). In general, `DBOS` with an online factor of 1.5 gets the best maximum stretch, followed by `DBOS` without the online factor. The difference between `DBOS` and `Iterative` is half an order of magnitude, and `SCOJO-P` is an order of magnitude worse than `DBOS`.



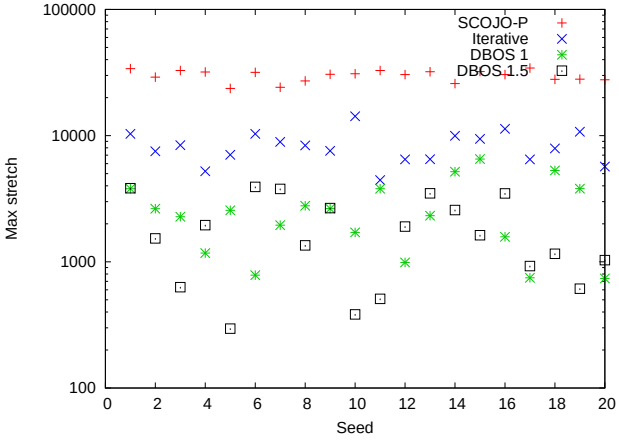
(a) Average Flow-Time



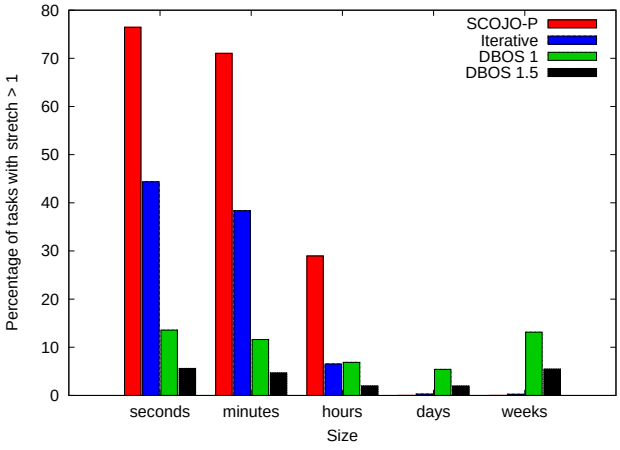
(b) Maximum Flow-Time



(c) Average Stretch

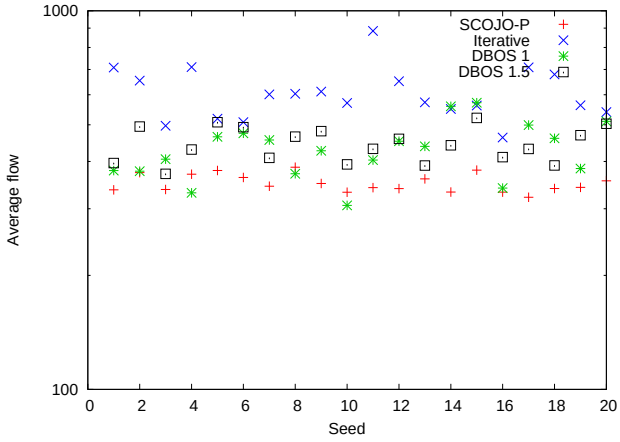


(d) Maximum Stretch

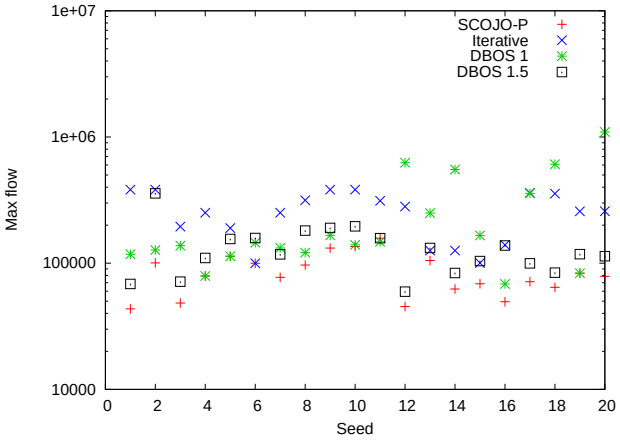


(e) Fairness Study

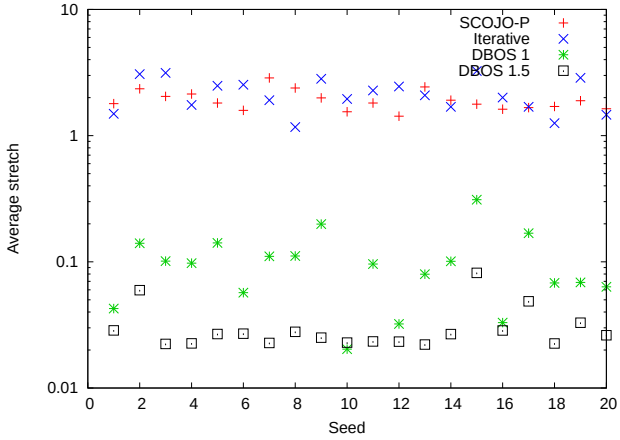
Figure 3: SDSC results.



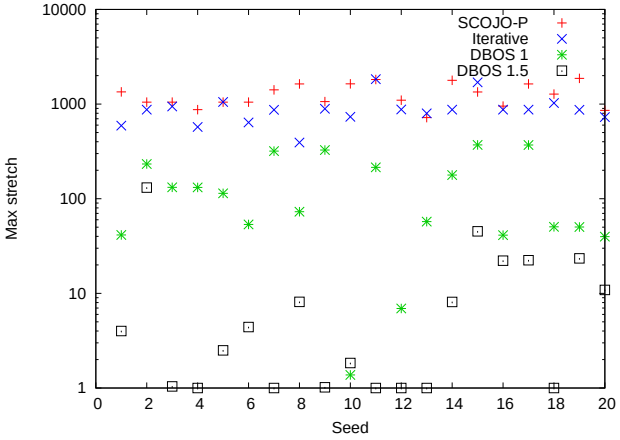
(a) Average Flow-Time



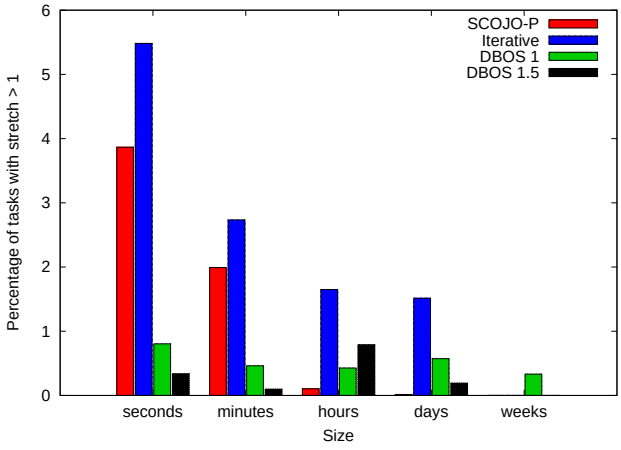
(b) Maximum Flow-Time



(c) Average Stretch



(d) Maximum Stretch



(e) Fairness Study

Figure 4: OSC results.

The histogram in Figure 3(e) presents the percentage of tasks that obtained a stretch larger 1 as function of their size for each of the considered algorithms. In this figure, the results from all 20 instances are aggregated. From left to right, the bins include tasks with sequential processing times expressed respectively in seconds, minutes, hours, days and weeks. The results show that the schedules created by `SCOJO-P` and `Iterative` are unfair against small tasks. When using `SCOJO-P`, more than 70% of the tasks shorter than an hour and 30% of the tasks shorter than a day obtain a stretch larger than one, whereas none of the tasks longer than a day obtains such stretch. The behavior of the `Iterative` algorithm is similar. More than 35% of the tasks shorter than an hour obtain a stretch larger than 1 whereas an insignificant portion of the tasks longer than a day gets such stretch. `DBOS`, especially when using the online factor, is found to be more fair in this respect. Around 3% of all the tasks get a stretch larger than 1 regardless of their size (the lowest is 1.96% for days and the highest is 5.6% for seconds). Recall that a task getting a stretch higher than 1 means that the scheduler failed to provide either a single processor to that task when it was submitted or sufficient moldability to compensate for it. Provided the load of the SDSC instance is 272, while there are 512 processors, Figure 3(e) shows that `SCOJO-P` and `Iterative` lead to inefficient schedules for small tasks

Figure 4 presents results similar to those in Figure 3 for the instances generated using the logs from the Ohio Supercomputing Center. The results in Figures 4(a) and 4(b) show that the flow-time performance of the algorithms on these instances are different from the corresponding results in Figure 3. `SCOJO-P` usually gets the best flow-time (in both average and maximum), `Iterative` gets the worst flow-time, and `DBOS` lies in between. In terms of average stretch (see Figure 4(c)), `DBOS` with an online parameter of 1.5 leads to two orders of magnitude better results than `SCOJO-P` and `Iterative`. Average stretch values as low as 0.03 for `DBOS` indicates that the parallelism of the tasks is utilized well. `DBOS` is also around 2 or 3 orders of magnitude better than `SCOJO-P` and `Iterative` in terms of maximum stretch (see Figure 4(d)). Moreover, the fairness of `DBOS` relative to `SCOJO-P` and `Iterative` is confirmed on the OSC data as well (Figure 4(e)). However, note that on OSC instances,

the percentage of tasks getting a stretch larger than 1 is never more than 6% whereas it can reach 75% on SDSC instances. We believe this difference in performance comes from OSC instances having a much smaller load than SDSC instances.

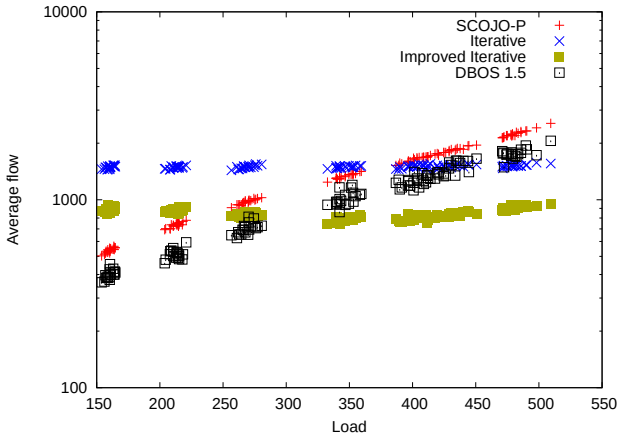
7.2. Results on instances from short sequence mapping application

In the second set of experiments, we consider workloads consisting of short sequence mapping tasks as described in the third scenario above. In these experiments, we also consider the modified version of the `Iterative` algorithm (the `Improved Iterative` algorithm described in Section 5.4) due to existence of steps in the speedup function of short sequence mapping tasks. The results are given in Figure 5. In the first four charts, the y-axis is in log scale and the results are presented as a function of load.

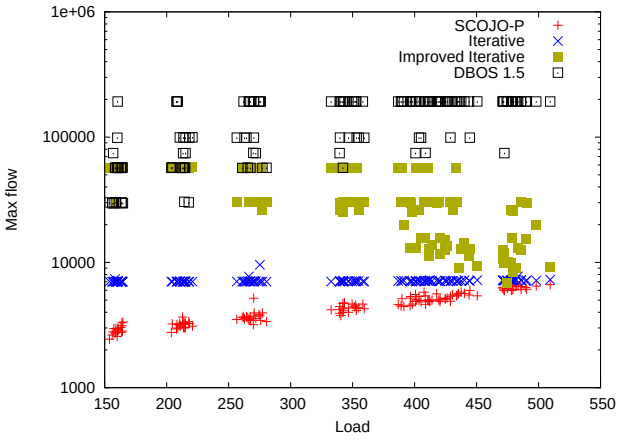
Surprisingly, the algorithms display different behaviors in average flow-time as a function of load (Figure 5(a)). The performance of `SCOJO-P` and `DBOS` degrades with increasing load, whereas the performance of `Iterative` and `Improved Iterative` stay almost constant. As expected, the `Improved Iterative` algorithm gets better average flow-times than the original `Iterative` algorithm in all instances. On the other hand, it is less expected that `DBOS` leads to better average flow-time than `SCOJO-P`. The best average flow-time is obtained by `DBOS` on instances with load smaller than 300 and by `Improved Iterative` on instances with load greater than 300.

In terms of maximum flow-time (Figure 5(b)), `SCOJO-P` performs the best among the considered algorithms. The second best algorithm is the `Iterative` algorithm which is significantly better than the `Improved Iterative` algorithm. The maximum flow-time of `DBOS` on the other hand, is around 1.5 order of magnitude worse than `SCOJO-P`. The difference between `Iterative` and `Improved Iterative` can easily be explained by the fact that the `Iterative` algorithm tends to use much less parallelism than the `Improved Iterative` algorithm. Therefore, its maximum flow-time is less dependent on the load. As the load increases, the parallelism utilized by `Improved Iterative` is also reduced, and its maximum flow-time converges to that of `Iterative`.

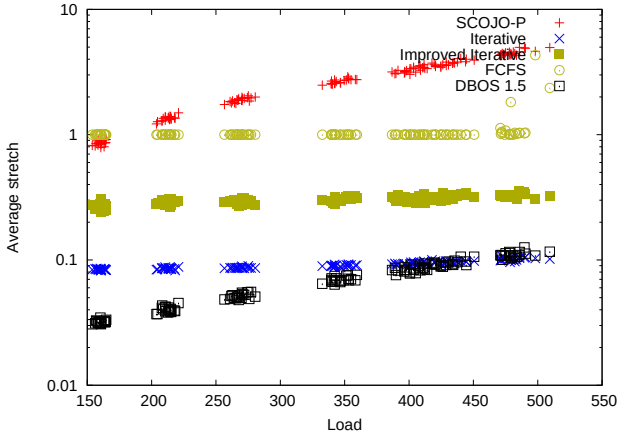
On average stretch (Figure 5(c)), the algorithm that leads to the best results is `DBOS` with



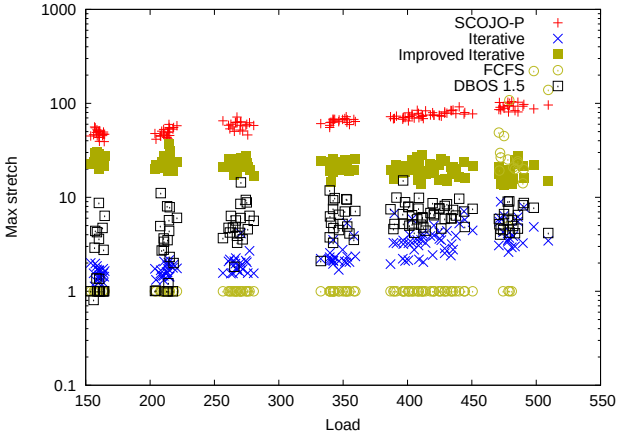
(a) Average Flow-Time



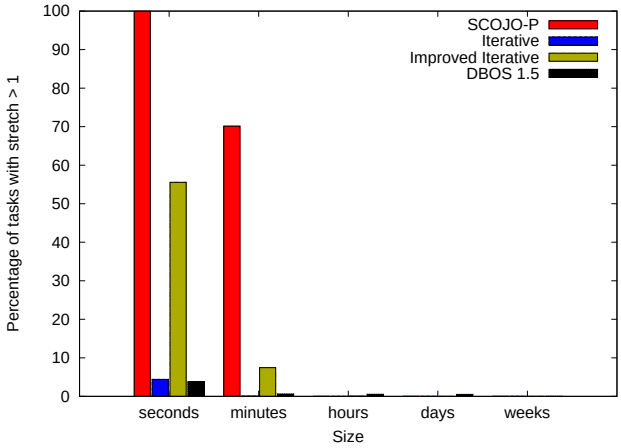
(b) Maximum Flow-Time



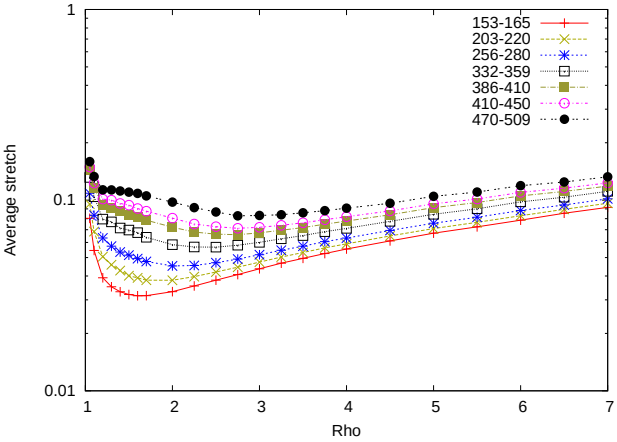
(c) Average Stretch



(d) Maximum Stretch



(e) Fairness Study



(f) Impact of ρ for various load

Figure 5: Short sequence mapping results.

stretch ranging from 0.025 to 0.1. Even under high load conditions, DBOS is able to utilize the parallelism of the tasks to obtain good speedup. SCOJO-P performs poorly and leads to average stretch greater than 1 for load values greater than 200, and average stretch greater than 5 for load values greater then 500. The Improved Iterative algorithm gets average stretch between 0.2 and 0.4, only slightly increasing with the load. The average stretch of the Iterative algorithm is steady around 0.1. Indeed, it can not get better stretch values since it is likely that the speedup function has a step around 11 or 13 processors.

Having a step in the speedup curve, however, is more interesting in terms of the maximum stretch for the Iterative algorithm (Figure 5(d)). Although not deliberately intended, failure to effectively utilize the available parallelism of the platform tends to leave more processors idle and increases availability for upcoming tasks. This results in worse performance in average flow-time (Figure 5(a)), but the Iterative algorithm results in very low maximum stretch in most cases.

The performance of the Iterative algorithm is a good example of the impact of machine availability on stretch. As a more extreme example, we provide the result obtained by a First Come, First Served (FCFS) scheme, where all tasks are executed sequentially to maximize machine availability. The results in Figures 5(c) and 5(d) indicate that when using FCFS all tasks have a stretch of 1 unless the load is too high. Although this leads to the best results for maximum stretch, the performance on average stretch becomes rather poor. On the other hand, DBOS achieves a better trade-off between maximum stretch and average stretch compared to Iterative and FCFS. The reason is that, similar to DASEDF for sequential tasks, the objective of DBOS is to minimize the instantaneous maximum stretch for moldable tasks. This results in higher processor utilization and keeps the average stretch low as well. The maximum stretch values obtained by DBOS are scattered between 1 and 10 whereas those obtained by the Improved Iterative algorithm are between 10 and 30. Note that, if the speedup curve of the short sequence mapping application was smoother, the performance of the Iterative algorithm would converge to the much worse performance obtained by the Improved Iterative algorithm. The maximum stretch of SCOJO-P is very high and further

degrades from 40 to 100 with increasing load.

The histogram in Figure 5(e) corresponds to those in Figures 3(e) and 4(e), and consists of tasks aggregated from the instances of load between 386 and 410. The results in this figure also attest that `SCOJO-P` and `Improved Iterative` generate unfair schedules against shorter tasks, whereas `DBOS` creates more fair and efficient schedules.

In the short sequence mapping application scenario, all the results of the `DBOS` algorithm have been presented with the online factor set to 1.5. To address the question of how to select a proper value for the online factor we present Figure 5(f), which shows the change in average stretch as a function of the online factor ρ under different load conditions. The performance of `DBOS` first quickly improves with increasing ρ, then it degrades with further increase. Furthermore, the optimal value for ρ increases with the load. The optimal value of ρ is 1.6 when the load is around 160, and it is 3 when the load is around 480. It is worth noting that the obtained average stretch varies only slightly around the optimal value of the online factor. Moreover, the average stretch seems to be bi-monotonic in ρ. Therefore, tuning this parameter should be easy in practice.

7.3. Runtime of the algorithms

The scheduling overheads of both `DBOS` and the `Iterative` algorithm are low and mainly depends on the number of tasks in the queue. On a regular desktop (2.4Ghz Intel Core2 processor, 2GB of memory), our implementation of `DBOS` takes about a minute to schedule 8000 tasks, while our implementation of `Iterative` takes a minute and a half. Even though a greedy algorithm would deliver the schedules faster, the computation times of these two algorithms are far from being prohibitive since the execution of tasks in a cluster can last for hours and since the scheduling process does not interfere with tasks already being executed. The runtime of `SCOJO-P` is much higher but because the `SCOJO-P` implementation was not built in the simulator the runtimes are not reported due to fairness concerns.

8. Conclusion

In this paper, we provide an analysis of stretch optimization problem for scheduling online non-preemptive tasks in a cluster environment. We first focus on the case of sequential tasks and propose a detailed theoretical analysis for two algorithms: `FCFS` and `DASEDF`. We show that machine availability is the key to optimizing stretch and the problem can be reduced to scheduling tasks with deadlines. By using the lessons learned from the study of sequential tasks we design the `DBOS` algorithm for optimizing stretch of moldable tasks.

`DBOS` is evaluated in comparison to two state-of-the-art scheduling algorithms: `SCOJO-P` and `Iterative`. We measure the average and maximum stretch as well as average and maximum flow-time under three different workload scenarios. `SCOJO-P` is found to reach the lowest maximum flow-times, whereas the best algorithm in terms of average flow-time is observed to vary according to the type of the workload and the load of the system. `DBOS` results in the best stretch values and is shown to be effective in minimizing both average and maximum stretch. Furthermore, by using a stretch-based objective function `DBOS` creates fair schedules with respect to task sizes.

As future work, the following tracks can be investigated. In maximum stretch optimization for sequential tasks, the gap between the approximation ratio of the best known algorithm and the best possible approximation ratio is large. We believe that an algorithm that reaches a $O(\frac{\Delta}{m})$ approximation ratio could be designed by looking for solutions that minimize the time between two successive completion times, visually similar to a staircase, if all the task have the same length. In scheduling moldable tasks, finding better ways of dealing with upcoming tasks can significantly reduce the maximum stretch.

Acknowledgment

This work was supported in parts by the U.S. DOE SciDAC Institute Grant DE-FC02-06ER2775; by the U.S. National Science Foundation under Grants CNS-0643969, OCI-0904809, and OCI-0904802.

We would like to thank Dr. A. Sodan for sharing with us an implementation of SCOJO-P.

References

[1] D. Jackson, Q. Snell, M. Clement, Core algorithms of the maui scheduler, in: D. G. Feitelson, L. Rudolph (Eds.), Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP), no. 2221 in LNCS, Springer, 2001.

[2] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, O. Richard, A batch scheduler with high level components, in: Proc. of Cluster Computing and Grid (CCGrid05), 2005.

[3] M. A. Bender, S. Chakrabarti, S. Muthukrishnan, Flow and stretch metrics for scheduling continuous job streams., in: Proc. of Symposium on Discrete Algorithms (SODA), 1998, pp. 270–279.

[4] J. Turek, J. L. Wolf, P. S. Yu, Approximate algorithms scheduling parallelizable tasks, in: Proc. of Symposium on Parallel Algorithms and Architectures (SPAA), ACM, 1992, pp. 323–332.

[5] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, P. Wong, Theory and practice in parallel job scheduling, in: Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS, Springer, 1997, pp. 1–34.

[6] D. Bozdag, C. C. Barbacioru, U. Catalyurek, Parallel short sequence mapping for high throughput genome sequencing, in: Proc. of International Parallel and Distributed Processing Symposium (IPDPS), 2009.

[7] K. Davies, Pacific Biosciences preparing the 15-minute genome by 2013, Bio IT World.

[8] B. Kalyanasundaram, K. Pruhs, Speed is as powerful as clairvoyance, Journal of ACM 47 (4) (2000) 617–643.

[9] E. Saule, D. Bozdag, U. Catalyurek, A moldable online scheduling algorithm and its application to parallel short sequence mapping, in: Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP), 2010.

[10] R. Graham, E. Lawler, J. Lenstra, A. R. Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Annals of Discrete Math. 5 (1979) 287–326.

[11] S. Muthukrishnan, R. Rajaraman, A. Shaheen, J. Gehrke, Online scheduling to minimize average stretch, in: Proc. of Foundation Of Computer Science(FOCS), 1999, pp. 433–443.

[12] A. Legrand, A. Su, F. Vivien, Minimizing the stretch when scheduling flows of divisible requests, Journal of Scheduling 11 (5) (2008) 381–404.

[13] X. Wu, V. C. S. Lee, Preemptive maximum stretch optimization scheduling for wireless on-demand data broadcast, in: Proc. of International Database Engineering and Applications Symposium, 2004, pp. 413–418.

[14] L. Becchetti, S. Leonardi, S. Muthukrishnan, Average stretch without migration, Journal of Computer and System Sciences 68 (1) (2004) 80–95.

[15] C. Chekuri, S. Khanna, A. Goel, A. Kumar, Multi-processor scheduling to minimize flow time with resource augmentation, in: Proc. of Symposium on Theory of Computing (STOC), 2004, pp. 363–372.

[16] N. Bansal, K. Dhamdhere, A. Sinha, Non-clairvoyant scheduling for minimizing mean slowdown, Algorithmica 40 (4) (2004) 305–318.

[17] A. Goel, M. R. Henzinger, S. Plotkin, E. Tardos, Scheduling data transfers in a network and the set scheduling problem, Journal on Algorithms 48 (2) (2003) 314–332.

[18] M. H. Bateni, L. Golab, M. T. Hajiaghayi, H. Karloff, Scheduling to minimize staleness and stretch in real-time data warehouses, in: Proc. of Symposium on Parallelism in Algorithms and Architectures (SPAA), 2009, pp. 29–38.

[19] K. Pruhs, J. Sgall, E. Torng, Online scheduling, in: J. Y.-T. Leung (Ed.), Handbook of Scheduling, 2004, Ch. 15.

[20] D. S. Hochbaum (Ed.), Approximation Algorithms for NP-hard problems, Course Technology, 1997.

[21] P. Brucker, Scheduling Algorithms, Springer-Verlag, 1995.

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

[23] C. A. Phillips, C. Stein, E. Torng, J. Wein, Optimal time-critical scheduling via resource augmentation, Algorithmica 32 (2) (2002) 163–200.

[24] S. Srinivasan, S. Krishnamoorthy, P. Sadayappan, A robust scheduling technology for moldable scheduling of parallel jobs, in: Proc. of Cluster, 2003, pp. 92–99.

[25] S. Srinivasan, R. Kettimuthu, V. Subramani, Selective reservation strategies for backfill job scheduling, in: Proc. of Job Scheduling Strategies for Parallel Processing, Springer-Verlag, 2002, pp. 55–71.

[26] K. Jansen, L. Porkolab, Linear-time approximation schemes for scheduling malleable parallel tasks, in: Proc. of Symposium on Discrete Algorithms (SODA), 1999, pp. 490–498.

[27] G. Mounie, C. Rapine, D. Trystram, A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks, SIAM Journal on Computing 37 (2) (2007) 401–412.

[28] P.-F. Dutot, L. Eyraud, G. Mounié, D. Trystram, Bi-criteria algorithm for scheduling jobs on cluster platforms, in: Proc. of Symposium on Parallel Architectures and Algorithms (SPAA), 2004, pp. 125–132.

[29] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, P. Sadayappan, Effective selection of partition sizes for moldable scheduling of parallel jobs, in: Proc. of High Performance Computing (HiPC), Vol. 2552 of LNCS, Springer, 2002, pp. 174–183.

[30] G. Sabin, M. Lang, P. Sadayappan, Moldable parallel job scheduling using job efficiency: An iterative approach, in: Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP), 2006, pp. 94–114.

[31] L. Barsanti, A. C. Sodan, Adaptive job scheduling via predictive job resource allocation, in: Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP), Springer Verlag, 2006, pp. 115–140.

[32] D. Feitelson, Parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload/.

[33] A. B. Downey, A parallel workload model and its implications for processor allocation, Cluster Computing 1 (1) (1998) 133–145.