A Moldable Online Scheduling Algorithm and Its Application to Parallel Short Sequence Mapping*

Erik Saule, Doruk Bozdağ, Umit V. Catalyurek
{esaule,bozdagd,umit}@bmi.osu.edu

Department of Biomedical Informatics, The Ohio State University

Abstract. A crucial step in DNA sequence analysis is mapping short sequences generated by next-generation instruments to a reference genome. In this paper, we focus on efficient online scheduling of multi-user parallel short sequence mapping queries on a multiprocessor system. With the availability of parallel execution models, the problem at hand becomes a moldable task scheduling problem where the number of processors needed to execute a task is determined by the scheduler. We propose an online scheduling algorithm to minimize the stretch of the tasks in the system. This metric provides improved fairness to small tasks compared to flow time metric and suits well to the nature of the problem. Experimental evaluation on two workload scenarios indicate that the algorithm results in significantly smaller stretch compared to a recent algorithm and it is more fair to small sized tasks.

1 Introduction

The rate of increase in DNA sequence information have greatly exceeded the expectations due to the emergence of next-generation sequencing instruments, including Roche's (454) GS FLX Genome Analyzer, Illumina's Solexa IG sequencer, and Applied Biosystem's SOLiD system, which are capable of sequencing more than one billion bases a day. The massive volumes of generated data pose new computational and analytical challenges that need to be addressed rapidly to keep up with the pace of the advancements in sequencing technology.

In many genome-wide and targeted studies, such as whole-genome resequencing, transcriptome analysis, small RNA analysis, targeted sequencing, DNA methylation and ChIP sequencing, one of the first steps to analyze the generated data sequences (reads) is to map them to a reference genome. This computationally intensive process involves mapping hundreds of millions of short reads generated in a typical run of a high throughput sequencing system to a reference genome that consists of up to three and a half billion bases. Since next

* 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, OCI-0904802 and CNS-0403342; and an allocation of computing time from the Ohio Supercomputer Center.

generation sequencing instruments usually generate reads as short as 35-50 base pairs, more specialized mapping algorithms such as MapReads [2], RMAP [21], MAQ [15], SOAPv2 [16] or Bowtie [13], have been introduced and shown to be more efficient than traditional local alignment algorithms BLAST, FASTA and their variants [1, 19, 26] for this particular problem. Even with these new algorithms, however, the mapping process takes days on a single computer which becomes a bottleneck in the application workflow given that the goal is to be able to sequence the entire human genome in 15 minutes by the year 2013 [5]. As a natural step to speed up the mapping process, several parallelization techniques have been proposed in our recent work [4] which apply to many short sequence mapping algorithms, i.e., those based on hashing or indexing the reference genome.

In this work, we consider online scheduling of multiple parallel short sequence mapping tasks in a multi-user environment. The methods introduced in [4] describe several ways of distributing the reads and the genome data onto the processors of a cluster to parallelize short sequence mapping process. Furthermore for each method, a cost model is provided to estimate the parallel execution time for a given number of reads and a given reference genome size. Using these cost models, it is possible to determine the best parallelization method and the estimated execution time for each short sequence mapping task on a given number of processors. Therefore, in the considered scheduling problem, the number of processors to be used for executing a task is decided by the scheduler based on the current load and availability in the system. In scheduling literature, such tasks are said to be *moldable* parallel tasks[1] as opposed to *rigid* parallel tasks which require the number of processors a task will use to be provided by the user.

In this paper, we propose an algorithm to schedule moldable tasks that arise in parallel short sequence mapping. Due to the large variety of task sizes and the availability of accurate execution time estimates, we focus on minimizing the *stretch* of the tasks in the system which is defined as the time a task spends in the system normalized by its execution time. Compared to the commonly used *flow time (average turnaround time)* metric, stretch provides fairness to all tasks in the system by including the execution time of the tasks in its definition. This objective was first studied for sequential tasks without preemption [3] and later with preemption [14] in the context of bag-of-tasks applications. To the best of the authors knowledge, this work is the first to consider the minimization of the stretch objective in moldable task scheduling without preemption.

The rest of this paper is organized as follows. In Section 2, we provide background information about parallel short sequence mapping. Sections 3 and 4 present moldable task scheduling and a brief discussion of two recent studies. We give details of the proposed scheduling algorithm for moldable tasks in Sec-

[1] They were originally called malleable tasks [25]. Feitelson *et al.* [9] made the distinction between constant number of processors and variable number of processors by using moldable for the former case and malleable for the latter one. However, they were still called malleable in more recent works.

tion 5. Then, we report results from our experimental studies in Section 6 and conclude in Section 7.

2 Parallel Short Sequence Mapping

The short sequence mapping problem asks for identifying the matching locations, possibly with some mismatches, of short input sequences (reads) on a reference genome. There are many mapping algorithms in the literature [2,13,15,16,21], most of which use a hash or an index table to store all consecutive same sized sub-sequences of either the reference genome or the query sequences to increase efficiency of the mapping process. The use of such data structures (i.e. hash or index), makes the problem less data dependent and enables accurate estimation of execution times by only taking global properties of the input problem into account.

A hashing based short sequence mapping algorithm consists of two major steps [4]. In the first step, a hash table is constructed by computing a hash value for each sub-sequence of the reference genome having length equal to read length. The execution time of this step can be modeled as $c_g G$, where c_g is the time needed to compute a hash value for a single sub-sequence and G is the size of the reference genome. In the second step, reads are matched to the genome by looking up their corresponding hash values in the hash table. When a fixed sized hash table is used, average number of collisions during table look-up depends on the number of entries in the table, which is proportional to genome size G. Therefore, the time required to process all reads can be modeled as $(c_r + c_c G)R$, where R is the number of reads, c_r is the constant work needed to process a single read, and c_c is a constant to capture additional work to resolve collisions. Then, the total execution time can be modeled as: $c_g G + (c_r + c_c G)R$.

As discussed in [4], straightforward methods to parallelize a mapping algorithm is to partition the reads and/or the reference genome to the processors of a cluster. This way, parallel execution time would be expressed as follows:

$$c_g \frac{G}{N_g} + (c_r + c_c \frac{G}{N_g}) \frac{R}{N_r} \tag{1}$$

where N_g and N_r respectively are the number of parts the genome and the reads are divided. For a cluster with m processors, $N_g \times N_r \leq m$ should be satisfied.

In addition to partitioning the reads and the genome, a new technique to assign reads and the genome to the processors is also introduced in [4]. In this method, called *Suffix Based Assignment (SBA)*, each processor is assigned a set of *suffixes* and is only held responsible for matching reads to the genome sequences that end in those suffixes. Each suffix consists of one or more nucleotide symbols. For example, if a processor is assigned the suffix AC, it is only responsible for matching reads that end in AC (e.g. ACCGTTA**AC**) to the genome sequences that also end in AC. Although SBA allows better parallelism, it comes with the cost of extra scan operations to compare sequences against the suffixes assigned to each processor. We represent the cost of checking the suffix of a genome and a read sequence by c_{gs} and c_{rs}, respectively. Moreover, we use N_s to denote the number of suffix groups to be considered. An example of suffix groups for $N_s = 2$ would be {A, C} and {G, T}. SBA can be applied in combination with reads and genome partitioning and can be considered as a new dimension for parallelism. Then, under perfect load balance, the parallel execution time can be formulated as follows

$$c_{gs}\frac{G}{N_g} + c_g\frac{G}{N_g N_s} + c_{rs}\frac{R}{N_r} + (c_r + c_c\frac{G}{N_g N_s})\frac{R}{N_r N_s} \tag{2}$$

where $N_g \times N_r \times N_s \leq m$ and $N_s > 1$. Remark that if $N_s = 1$, there is no need to do suffix checking, hence with $c_{gs} = 0$ and $c_{rs} = 0$ Equation (2) reduces to Equation (1).

Please note that, by using tree-based one-to-all data distribution scheme, data distribution time, which includes distribution of input sequences and possibly reference genome to the processors of the parallel machine, becomes negligible in comparison to actual mapping computations. Therefore, it is omitted in these formulas. Furthermore, our earlier work [4] shows that these estimates are accurate.

3 Moldable Task Scheduling

3.1 Problem formulation and properties

In this section we discuss details about online scheduling of parallel short sequencing mapping tasks in a multi-user environment. We consider a typical online setting, where n independent tasks are dynamically submitted to a cluster of m identical processors. Arrival time of task i to the system is denoted by r_i. We use the notation $p_{i,j}$ to represent the execution time of task i on j processors. Information about arrival or execution time of the tasks are not available to the scheduler until submission. The scheduling problem we consider is to decide the number of processors π_i to be allocated for each task i and the time σ_i when the execution of task will start on the system. The completion time C_i of task i is $C_i = \sigma_i + p_{i,\pi_i}$.

In the short sequence mapping problem considered in this paper, each task i corresponds to a mapping request of R_i reads on a genome of size G_i. Therefore, for each task i and each number of processors $j \leq m$, $p_{i,j}$ is computed using Equation (2) by replacing G with G_i and R with R_i. The values for N_r, N_g and N_s are chosen such that the total execution time predicted by this equation is minimized for the given values of G_i, R_i and j. In short sequence mapping, storing the genome in a hash table implies high memory requirement that prevents two tasks to be executed simultaneously on the same processor. Thus, preemption is not allowed. *Monotony* of computation time and absence of super-linear speedup are common assumptions in moldable scheduling. One can check that they are valid for the parallel short sequence mapping tasks.

3.2 Objective functions

The general objective in online scheduling is to execute all submitted tasks without delaying their execution too much in the system. A desired property of a scheduler is to avoid starvation while ensuring an overall good response time.

The most studied objective functions in moldable task scheduling are based on aggregation of completion time (makespan) of the tasks, such as the minimization of maximum completion time and minimization of average completion time. A common technique to optimize completion time is to use dual-approximation [12, 17]. This technique consists of choosing a target value for the objective and then to decide 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.

A commonly used metric in online scheduling that does not require a time origin is the *flow time* (also known as turnaround time). Flow time of a task i is the time the task spends in the system and is calculated as $F_i = C_i - r_i$. Two related objective functions are the minimization of maximum flow time ($F_{max} = \max_i F_i$) and the minimization of the average flow time ($\frac{\sum_i F_i}{n}$). The former is especially well known for preventing starvation and is usually optimized by using the first-come first-serve (FCFS) ordering. Examples of flow minimization can be found in [7, 20, 24]

Since the flow time metric does not take the size of the tasks into account, objective functions that utilize this metric tend to create schedules in which small tasks spend as much time in the system as the large tasks. This results in small tasks waiting in the system queue longer than the large tasks, hence introduces unfairness against small tasks. To avoid this situation, the *stretch* metric can be used to replace flow time in the objective function. The stretch of a task is defined as the time spent by the task in the system normalized by its processing time. This metric has been studied for online scheduling of rigid tasks with preemption in [18] and for sequential task scheduling without preemption in [3] and then with preemption in [14]. However, to the best of the authors' knowledge, it has never been used nor defined in online moldable scheduling without preemption. We use the processing time of the task on one processor for normalization and the stretch of a task i is $s_i = \frac{C_i - r_i}{p_{i,1}}$. Corresponding objective functions for stretch are the minimization of maximum stretch ($S = \max_i s_i$) and the average stretch ($\frac{\sum s_i}{n}$).

Using an adversary technique, one can prove that in an online setting it is not possible to get an approximation algorithm for the minimization of maximum or average stretch objectives without preemption even if the system is composed of a single processor. Adversary technique works by dynamically constructing the instance that worsens the performance the most by taking advantage of the decisions of the algorithm. In this case, the idea is to first feed one long task to the scheduler. Once the execution of that task starts, the adversary submits a bunch of much smaller tasks to the system. Since these small tasks cannot start before the execution of the first task completes, the stretches of the small tasks are as large as the ratio between the execution time of the smallest task to that of the largest task. Such analysis is usually only useful for designing approximation algorithms. However, Section 6 indicates that similar effects also appear in practice.

In the job 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 actual execution time of the task. As a result, the slowdown always greater than one. Stretch can be considered as an extension of slowdown for the moldable tasks model.

A variant of the slowdown objective function is Bounded slowdown (BSLD) [10], which is used to avoid over-emphasizing the significance of small tasks. In this objective function, the processing time of the tasks are assumed to be greater than a given constant. Since this may resut in some taks 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 [23], which is defined as $\frac{queuingtime+p_{i,1}}{p_{i,1}}$. Xfactor is always greater than one and it does not take into account the number of processors used to execute a task.

3.3 Backfilling strategies

In most scheduling algorithms, tasks are scheduled as soon as possible in the order of their arrival times. This approach tends to create holes in the schedule, which can later be utilized 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, the task in conflict that has the largest start time is rescheduled. This approach provides a much 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. More details on backfilling strategies can be found in [22].

4 Analysis of Existing Solutions

4.1 The fair-share scheduling algorithm

The fair-share scheduling algorithm has been proposed in [24] and refined in [20] to optimize the average turnaround time. The basic principle of the original algorithm is to greedily schedule tasks one by one to minimize their completion time using aggressive backfilling. However, this approach leads to scheduling each task to execute in parallel using all processors in the system. Since efficiency usually decreases with the number of processors, tasks spend too much time in the system using this approach. To avoid such scenarios, 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 first 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 better results. However, this value was reported to be too restrictive and multiplied by an overbooking factor to allow the scheduler to consider a larger number of possibilities [20].

The fair-share algorithm induces starvation due to aggressive backfilling which can delay all the tasks but the first to be executed. Therefore, the tasks are partitioned in multiple queues based on the their sizes. Ensuring that the first task of each queue is never delayed reduces starvation. To further reduce starvation, the Xfactor of a task is introduced: $\text{Xfactor}(i) = \frac{t+p_{i,1}-r_i}{p_{i,1}}$, where t is the current time. A task whose Xfactor exceeds a certain threshold is no longer allowed to be rescheduled by the aggressive backfilling technique.

4.2 Iterative moldable scheduling algorithm

The fair-share algorithm provides fairly good performance but requires tuning many parameters. In [20], 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, i.e., decide the number of processors to be allocated for each task. Then, the tasks are scheduled using a conservative backfilling technique. The order in which the task are considered for backfilling is not given in [20]. In the following we assume that tasks are considered in the FCFS order.

The question of how many processors to allocate for each task is addressed using a simple principle. The algorithm starts by allocating one processor to each task and computing the corresponding schedule. Then, the task that would have the most reduction in its processing time by using an extra processor is found. Subsequently, an additional processor is assigned to that task and a new schedule is computed. If the new schedule has a better average turnaround 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 this 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. Similarly, in the short sequence mapping problem the values of N_r, N_g and N_s in Equation (2) have to be integer. If the number of processors m is prime, it is likely that the optimal partitioning scheme uses at most $m-1$ processors, which induces steps in the speedup function.

Improvements to the algorithm: Existence of steps in the speedup function results in early termination of the iterative scheme in the algorithm of Sabin *et al.* [20]. 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 speed up function is not convex, the modification allows to skip the allocation sizes that would lead to low efficiency (and thus skips steps). Throughout the paper, we refer to this variant of the algorithm as the *improved iterative* algorithm.

5 Deadline Based Online Scheduling

In this work, we propose an algorithm called *Deadline Based Online Scheduling (DBOS)* with the goal of minimizing the stretch of the tasks in the system. Throughout the section, we consider a typical system where the scheduler is invoked when a task enters or exits the system. Using the `DBOS` algorithm, the scheduler computes a new schedule for all tasks pending in the system queue. Tasks that have already started execution are kept running.

The outline of the `DBOS` algorithm is presented in Algorithm 1. The main idea in the algorithm is to compute the "best" achievable maximum stretch, denoted by S, using a binary search within lines 2–14. At each iteration of the binary search, a new schedule is computed by calling the `MoldableEDF` (for Moldable Earliest Deadline First) procedure using the current value of S. If the returned schedule is not feasible, S is increased. Otherwise it is decreased to find a tighter bound for maximum stretch. Since there is no apriori upper bound on S, the algorithm starts with computing one in lines 2–6.

Once the "best" feasible value of S is found, it is multiplied by an online factor ρ. The reason for relaxing the S value is to increase the efficiency of the system as well as to leave potentially more processors to the tasks that will arrive in the future. Furthermore, this helps improving the performance in the adversary scenario discussed in Section 3.2. The online factor ρ is the key to the online aspect of the `DBOS` algorithm.

In lines 20–30 of Algorithm 1, the details of the `MoldableEDF` procedure is given. Given a value of S, `MoldableEDF` starts by computing a deadline $D_i = r_i + p_{i,1}S$ for each task i (lines 21–22). This reduces the problem to scheduling the tasks before their deadlines. Then, the tasks are scheduled greedily in non-decreasing order of their deadlines. 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

Algorithm 1 Deadline Based Online Scheduling Algorithm

```
1:  procedure DBOS(INPUT: $\rho$, OUTPUT: $\pi^*$, $\sigma^*$)
2:      $LB \leftarrow 0$, $S \leftarrow 1$
3:      while Not Feasible $(\pi, \sigma)$ do        ▷ Compute an initial feasible maximum stretch
4:          $S \leftarrow 2S$
5:          $(\pi, \sigma) \leftarrow MoldableEDF(S)$
6:      $UB \leftarrow S$
7:      while $UB \neq LB$ do        ▷ Find the best maximum stretch using a binary search
8:          $S \leftarrow \frac{UB+LB}{2}$
9:          $(\pi, \sigma) \leftarrow MoldableEDF(S)$
10:         if Feasible $(\pi, \sigma)$ then
11:             $(\pi^*, \sigma^*) \leftarrow (\pi, \sigma)$
12:             $UB \leftarrow S$
13:         else
14:             $LB \leftarrow S$
15:     $(\pi^\rho, \sigma^\rho) \leftarrow MoldableEDF(\rho S)$        ▷ Relax $S$ by a factor of $\rho$ if it is feasible
16:     if Feasible $(\pi^\rho, \sigma^\rho)$ then
17:         $(\pi^*, \sigma^*) \leftarrow (\pi^\rho, \sigma^\rho)$
18:     return $(\pi^*, \sigma^*)$
19:
20: procedure MoldableEDF($S$)
21:     for all $i \leq n$ do        ▷ Compute a deadline for each task
22:         $D_i \leftarrow r_i + p_{i,1} S$
23:     Construct initial processor allocation using information about running tasks
24:     for all task $i$ in non-decreasing $D_i$ order do
25:         for all $j$ from 1 to $m$ do
26:             $x \leftarrow$ earliest time that $j$ processors are available for $p_{i,j}$ units of time
27:             if $x + p_{i,j} \leq D_i$ then
28:                 $\pi_i \leftarrow j$; $\sigma_i \leftarrow x$
29:                 Exit inner for loop
30:     return $(\pi, \sigma)$
```

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 23 to line 29 is generic. It could be used for a classical scheduling problem of moldable tasks with deadline.

The algorithm has several interesting properties. First of all, if `MoldableEDF` was an exact algorithm, then the optimal maximum stretch would be found. The deadline scheduling problem as well as the maximum stretch optimization problem are NP-Complete [11]. However, it is likely that if an approximation algorithm for the deadline scheduling problem was known, it would lead to an approximation algorithm for the maximum stretch optimization problem.

`MoldableEDF` is a greedy algorithm and is not optimal as it can fail to find the best feasible solution. However, the `MoldableEDF` is based on two principles that make it efficient. First, the tasks are considered in non-decreasing order of deadlines. This principle, called Earliest Deadline First, leads to optimality in single processor deadline scheduling problems and provides guaranteed approx-

Sequencing machine	Number of reads
454 GS FLX Genome Analyzer	1 million
Solexa IG sequencer	200 million
SOLiD system	400 million

Genome	Size (bases)
E. Coli	4.6 million
Yeast	15 million
A. Thaliana	100 million
Mosquito	280 million
Rice	465 million
Chicken	1.2 billion
Human	3.4 billion

Table 1. (Left) Sequencing machines and the number of reads each of them produces in a single run. (Right) Genomes and their sizes.

imation for the sequential task scheduling problem on an arbitrary number of processors. Second, the algorithm allocates the minimum number of processors that ensures a task matches its deadline. This decision maximizes the processor availability for the other tasks in the system, hence helps keeping the system efficiency high. Moreover, it helps avoiding local optima due to presence of steps in the speedup function. This principle is similar to the canonical number of processors used in makespan optimization.

6 Experiments

Execution time of short sequence mapping tasks vary significantly depending on the size of the reference genome and the number of reads to be mapped (see Table 1). For instance, a targeted sequence analysis involves mapping a few million reads 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 reads and may take a few hours for mosquito and a few days for human genome. In this section, we report on the simulation results of the DBOS algorithm on a 512-processor cluster using two workload scenarios that reflect such variety in task execution times. The first scenario is based on a log file from a supercomputing center and is included to assess the performance of the algorithm on well known data. The second scenario is designed to simultate the load of a cluster dedicated for short sequence mapping tasks.

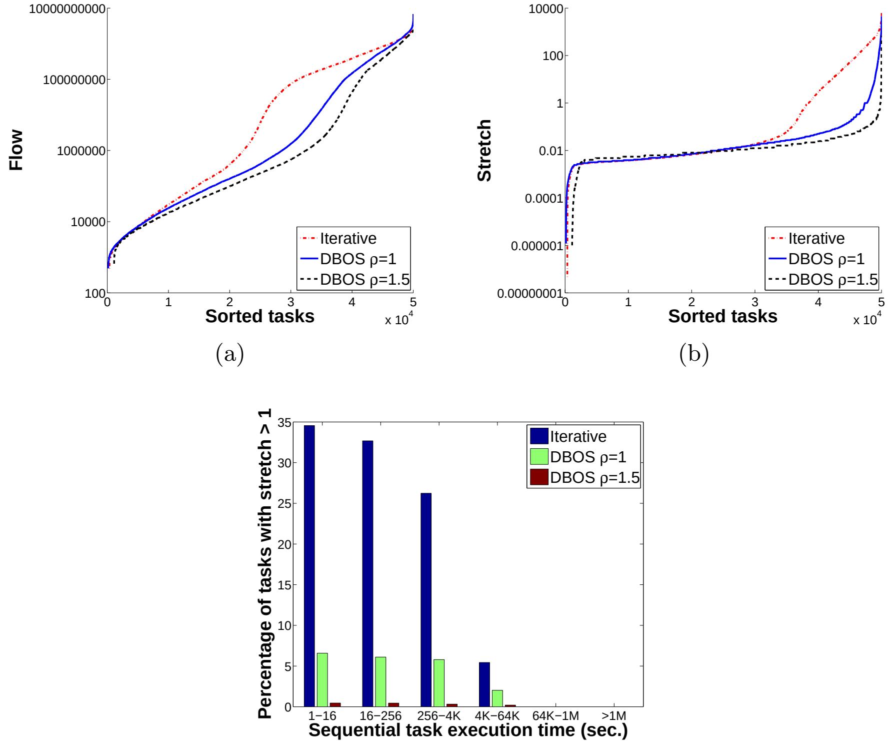
In the first scenario, we used a real log file (SDSC Par 96 in [8]) of parallel jobs submitted to the San Diego Supercomputing Center (SDSC). This file contains information about a task's arrival time, runtime on the system and the number of processors used for its execution. We considered the first $5,000$ tasks, and similar to [20], we used the Downey model [6] 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 [6]. Since the Downey model is stochastic, 10 different instances were generated.

In the second scenario, each workload consists of 5,000 parallel short sequence mapping tasks and each task arrives at the cluster with an inter-arrival time chosen from a exponential distribution of parameter λ_i. We varied λ_i to obtain 6 different load conditions, where load is defined as the ratio of the sum of sequential processing times of all tasks to the time that elapsed between the arrival of the first and the last task. In other words, for a load of l, if all tasks were executed sequentially, then 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. Therefore, in our tests, if the load is larger than 512, the cluster is clearly overloaded. However, due to non-linear scalability of the tasks and random arrival times, it is very likely that the cluster gets overloaded even for load values less than 512. A task in this scenario represents a mapping operation of short sequences generated by one of the sequencing machines to one of the genomes listed in Table 1. The sequencing machine and the genome associated with a task is chosen randomly and the parallel execution time of the task is computed using the formulas from Section 2. The sequential processing time of the generated tasks vary between 30 seconds and 22 days.

Results of the DBOS algorithm are presented in comparison to the iterative algorithm of Sabin *et al.* [20] which was described in Section 4.2.

6.1 Downey model

First we present aggregate results from 10 runs using Downey model on the SDSC log file. Since 5,000 tasks are scheduled in each run, we had scheduling information about 50,000 tasks in 10 runs. In Figure 1(a), the flow time of these 50,000 tasks are shown in increasing flow time order for DBOS and the iterative algorithms (the improved version of the iterative algorithm is not presented here as it is equivalant to the original iterative algorithm since there are no steps in speedup functions of the Downey model). Figure 1(b) shows the corresponding chart with stretch on the y-axis. Due to wide variation of flow times and stretch values, log scale is used in the y-axis of both charts. These results show that on the average DBOS provides a better flow time and stretch compared to the iterative algorithm. Recall that if a task has a stretch greater than 1, it means that the time it spends in the system is greater than its sequential execution time. In other words, the speedup gain due to parallel execution is lost. The iterative algorithm resulted in more than 23% of the tasks to have stretch greater than 1, whereas the corresponding quantity was only 6% for DBOS with $\rho = 1$. The results improved even further when the value of ρ is increased to 1.5. In that case, only 1% of the tasks had a stretch greater than 1. In Figure 1(c), the percentage of tasks with stretch greater than 1 is shown for different task-size groups. The results indicate that the iterative algorithm results in a relatively unfair schedule by penalizing smaller tasks more in terms of their stretch. For example, 34% of the tasks in the smallest task-size group have a stretch larger than 1. DBOS results in a more fair schedule, where less than 7% and 1% of the



(a)

(b)

(c)

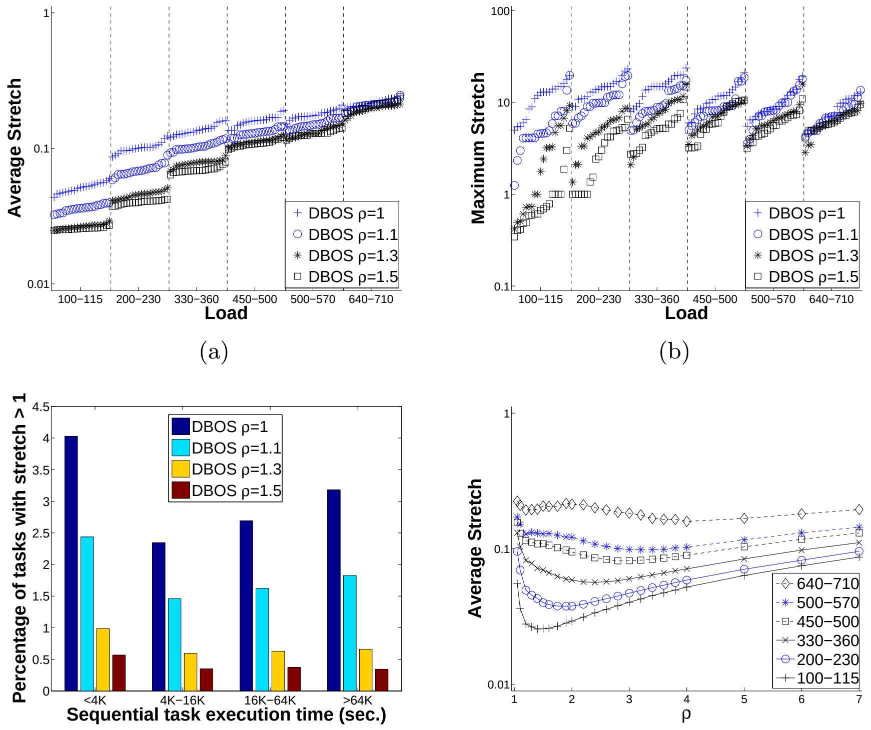
Fig. 1. Comparison of DBOS and the iterative algorithm on the SDSC/Downey workload. The y-axis is in log scale for (a) and (b). The lower is the better in all figures.

tasks had a stretch greater than 1 even for the smallest tasks with $\rho = 1$ and $\rho = 1.5$, respectively.

Note that larger tasks can afford longer delays without much degradation in their stretch. However, smaller jobs suffer more especially when the cluster is overloaded. This is similar to the worst case online scenario on a single processor as mentioned in Section 3.2, in which a very short task arrives just after a very long task is scheduled. Existence of some tasks getting a stretch over 100 in our experiments is the proof that such phenomenon appears also in practice. Nevertheless, if a value larger than 1 is used for the online factor ρ this behavior occurs rarely.

6.2 The Short Sequence Mapping Application

In the second set of experiments we considered workloads consisting of short sequence mapping tasks as described in the second scenario above. We generated 6 different load cases and for each case we generated 20 workloads. Since



(a)

(b)

(c)

(d)

Fig. 2. Impact of the online factor ρ on the performance. The y-axis is in log scale for (a), (b) and (d). The lower is the better in all figures.

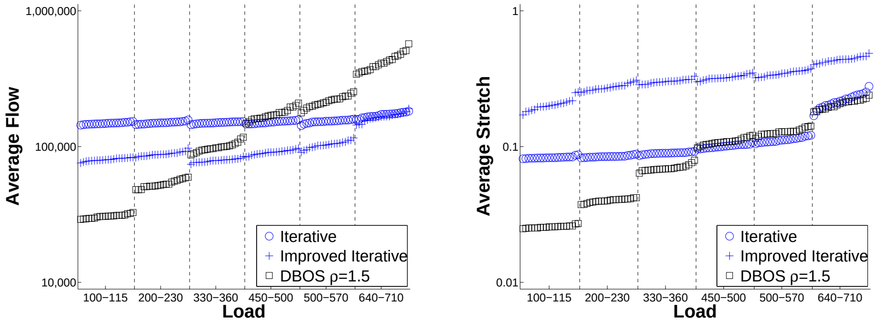
the instance generation will not provide the same load when run with the same parameters, we considered a range of load values around a targeted load value (and tuned the λ_i parameter to reach this load). The 6 load cases in the experiments correspond to the following ranges of load values: 100-115, 200-230, 330-360, 150-500, 500-570 and 640-710. Note that in the last two cases the cluster is overloaded (as the load is greater than the number of processors). These cases are included to see the performance of the algorithm in extreme load conditions.

We started with assessing the impact of the online factor ρ of the `DBOS` algorithm and used values of ρ chosen from the set $\{1, 1.1, 1.3, 1.5\}$. Recall that the parameter ρ allows to take the online characteristics of the problem into account by relaxing the instant maximal stretch to improve overall efficiency. In Figure 2(a), average stretch achieved by the `DBOS` algorithm with different ρ values are given under different load cases. For each load case and for each ρ value, the average stretch values are shown sorted. Figure 2(b) displays the corresponding results for maximum stretch values. In Figure 2(c), the percentage of tasks with stretch greater than 1 is shown for different task-size groups using the aggregate results from all 20 workloads that have load in the 330-360 range.

The results in Figure 2 suggest that both average and maximum stretch improves significantly with ρ until $\rho = 1.3$, after which the improvement is marginal. In general, using a ρ value greater than 1 results in an increase in the stretch of the tasks with extremely small stretch and a decrease in the stretch of the tasks with extremely large stretch (results were similar to those in Figure 1(b), hence omitted). Therefore, using larger ρ values helps reducing the variance of stretch as well as the average and maximum stretch. As seen in Figure 2(c), small sized tasks benefit the most from larger ρ values, as they are more likely to get large stretch values due to cases similar to the worst-case scenario described in Section 3.2. As the load in the system increases, there are far more tasks in the system and the online factor becomes less effective as it is no longer sufficient to keep a portion of the processors available for the tasks that will arrive in the future. Figures 2(a) and 2(b) illustrate that the online factor has very little impact in the two extreme load cases, where load is greater than 500.

In order to determine a reasonable range of values for the online parameter ρ, we computed the average stretch for different ρ values under different load conditions. The results of this experiment are given in Figure 2(d), where each point is the average over 20 instances of similar loads. Since the variance of average stretch values is low, (see Figure 2(a)), the standard deviation is omitted in this figure for clarity. The results show that the optimal value of ρ depends on the load of the system. The average stretch quickly drops when ρ increases as more room is created for small tasks. Then it slowly increases as all the tasks get delayed and some machines of the cluster are left idle. The shape of the curve allows easy estimation of the optimal ρ with a gradient method. Moreover, note that the average stretch has small variation around the optimal ρ value. For instance, for a load between 330 to 360, the optimal ρ value is 2.4 and all ρ values between 1.6 and 3.8 result in average stretch values within 20% of the optimal. Therefore, fine tuning of the ρ value is not essential as long as unreasonable values are avoided. In the rest of the experiments, the value of ρ is set to 1.5 which is a reasonable value for underloaded cluster scenarios.

In Figure 3 the results of the `DBOS` algorithm on short sequence mapping workloads are presented in comparison to the two variants of the iterative algorithm mentioned in Section 4.2: the original algorithm in [20], and the improved version for non-convex speedup function. These two variants lead to different results due to steps in the speedup curves of the short sequence mapping tasks. Results in Figure 3(a) show that the improved version leads to around 50% improvement in flow time; steps in speedup curves prevent the original version from using available parallelism. On the overloaded cases, the two versions are comparable since there are more tasks in the queue and both algorithms use all available processors. If the average flow time is the target metric in an application, our proposed improvement should be used in the iterative algorithm to handle non-convex speedup curves.



(a) (b)

Fig. 3. Comparison of DBOS and the iterative algorithm on short sequence mapping application workloads.

However, under-utilization of the cluster in the original iterative algorithm results in better stretch values. Indeed, the improved algorithm tends to utilize all processors in the cluster, thus tasks entering the system are delayed and get large stretch values. The original version results in many processors to remain idle, therefore, tasks that enter the system are scheduled immediately and obtain small stretch values. However, note that if the application had required the number of processors to be a power of two, the original iterative algorithm would never schedule a task on more than two processors, hence would not get a stretch better than 0.5. This is worse than the improved version which reaches an average stretch of 0.3. On the other hand, if the first step in the speedup curve appears on a large number of processors, the behavior of the original iterative algorithm converges to that of the improved one.

As clearly seen in Figure 3, DBOS achieves better stretch than both variants of the iterative algorithm. The difference is especially larger for low load conditions, where more than 70% improvement is achieved relative to the original iterative algorithm. The performance of the original iterative algorithm is comparable with DBOS only under the cases where the cluster had a load greater than 400. DBOS outperforms the revision of the iterative algorithm up to 85% on low load cases.

In terms of flow time, DBOS achieves better results than the iterative algorithm under low and medium load and worse results only in overloaded cluster conditions. Results in Figure 3 leads to the conclusion that DBOS achieves a balance between inefficient over-parallelism as in the case of improved iterative algorithm, and under-utilization of the cluster as in the case of original iterative algorithm. Therefore, except for extreme load conditions, it usually gives the best stretch and flow time among the considered algorithms.

The scheduling overhead 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 unoptimized implementation of DBOS and the iterative algorithm take about 20 to 30 seconds to schedule 5000 tasks. Despite a greedy algorithm would deliver the schedules faster, the computation times of the benchmarked 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.

7 Conclusion

The most computationally demanding step in DNA sequence analysis is mapping sequences generated by next-generation sequencing instruments to a reference genome. In this paper, we investigated online scheduling of multiple parallel short sequence mapping tasks in a multi-user environment. Availability of accurate estimates of parallel execution times of short sequence mapping queries allows using the moldable task model in the scheduling process. Existing studies mainly focus on optimizing the average flow time of tasks, which produces schedules unfair against small tasks. In the context of sequential tasks, one of the proposed solutions to address the fairness issue was the use of stretch metric. To the best of our knowledge, the work presented in this paper is the first that uses the stretch metric for moldable task scheduling without preemption. Experiments on two different workload scenarios, one based on the log of a production batch system and one reflecting realistic use-case scenario of the short sequence mapping application, showed that the proposed DBOS algorithm provides better schedules than the compared algorithms in terms of the stretch metric while improving the flow time on many cases. The results demonstrated that DBOS achieves a balance between inefficient over-parallelism and under-utilization of the cluster, two competing issues regarding online task scheduling.

References

1. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
2. Applied Biosystems. *MapReads: SOLiD System Color Space Mapping Tool.* `http://solidsoftwaretools.com/gf/project/mapreads/`.
3. M. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *Proc. of the Symposium on Discrete Algorithms*, pages 762–771, 2002.
4. D. Bozdag, C. C. Barbacioru, and U. Catalyurek. Parallel short sequence mapping for high throughput genome sequencing. In *Proc. of the International Parallel and Distributed Processing Symposium*, May 2009.
5. K. Davies. Pacific Biosciences preparing the 15-minute genome by 2013. *Bio IT World*, Feb. 2008.
6. A. B. Downey. A parallel workload model and its implications for processor allocation. *Cluster Computing*, 1(1):133–145, 1998.
7. M. Drozdowski and P. Dell'Olmo. Scheduling multiprocessor tasks for mean flow time criterion. *Computers and Operations Research*, 27(6):571–585, 200.
8. D. Feitelson. Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

9. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Proc. of Job Scheduling Strategies for Parallel Processing*, LNCS, pages 1–34. Springer, 1997.
10. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lect. Notes Comput. Sci.*, pages 1–34, 1997.
11. M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, 1979.
12. K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *proc. of 10th SODA*, pages 490–498, 1999.
13. B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25+, 2009.
14. A. Legrand, A. Su, and F. Vivien. Minimizing the stretch when scheduling flows of biological requests. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures*, 2006.
15. H. Li, J. Ruan, and R. Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, November 2008.
16. R. Li, C. Yu, Y. Li, T.-W. W. Lam, S.-M. M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics (Oxford, England)*, 25(15):1966–1967, August 2009.
17. G. Mounie, C. Rapine, and D. Trystram. A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM J. Comput.*, 37(2):401–412, 2007.
18. S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. Gehrke. Online scheduling to minimize average stretch. In *Proc. of FOCS*, pages 433–443, 1999.
19. W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. National Academy of Sciences*, 85:2444–2448, 1988.
20. G. Sabin, M. Lang, and P. Sadayappan. Moldable parallel job scheduling using job efficiency: An iterative approach. In *Proc of the 12th JSSPP*, pages 94–114, jun 2006.
21. A. D. Smith, Z. Xuan, and M. Q. Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC bioinformatics*, 9(1):128+, February 2008.
22. S. Srinivasan, R. Kettimuthu, and V. Subramani. Selective reservation strategies for backfill job scheduling. In *Job Scheduling Strategies for Parallel Processing, LNCS 2357*, pages 55–71. Springer-Verlag, 2002.
23. S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling technology for moldable scheduling of parallel jobs. In *Proc. of Cluster 2003*, pages 92–99, 2003.
24. S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Effective selection of partition sizes for moldable scheduling of parallel jobs. In *Proc. of the 9th HiPC Conference*, volume 2552 of *LNCS*, pages 174–183. Springer, 2002.
25. J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proc. of the fourth Symposium on Parallel Algorithms and Architectures*, pages 323–332. ACM, 1992.
26. Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7(1/2):203–214, 2000.