Scheduling Jobs with Non-uniform Demands on Multiple Servers without Interruption

Sungjin Im, Mina Naghshnejad and Mukesh Singhal
Electrical Engineering and Computer Engineering, University of California, Merced, CA 95343
Email: {sim3, mnaghshnejad, msinghal}@ucmerced.edu

***Abstract*—We consider the problem of scheduling jobs with varying demands on multiple servers. Each server has a certain computing capacity and can schedule multiple jobs simultaneously as long as the jobs' total demand does not exceed the server's capacity. This scenario arises commonly in virtualization, cloud computing, and MapReduce (or Hadoop). We study this problem with the requirement that jobs must be scheduled non-preemptively, meaning that every job must be completed without interruption once it gets started. Often, preemption is out of choice since preempting a job can be prohibitively costly or is not allowed due to system constraints. We focus on the popular objective of minimizing total completion time of jobs. This problem is NP hard hence we study heuristics with provable approximation guarantees. Succinctly, the interaction between two orthogonal quantities, jobs demands and sizes makes the scheduling decision significantly more challenging.**

In this paper we propose novel algorithms for scheduling jobs with non-uniform demands on multiple homogeneous servers without preemption. We first observe that the Smallest Volume First (SVF) algorithm that favors jobs with smaller volumes could perform very poorly in general. However, we show that SVF yields a *nearly optimal* schedule when the system is overloaded and jobs have demands considerably smaller than servers' capacities. This result supports the intuition that SVF should work well unless some jobs with high demands occupy the servers for long, blocking other jobs. Building on this intuition and using reduction to geometric packing problems, we develop algorithms that are constant approximation for *all* instances for the *first* time. Prior to our work, there was no theoretical study on this problem even for the single server case.

I. Introduction

Modern data centers consist of a large number of servers to handle explosive growth of data. Further, each server is getting increasingly powerful with more resources. For example, multi-processor chips have become dominant as the uniprocessor chip design has hit the thermal wall by producing too much heat to be cooled down economically. It is widely expected that more processors be packed into a single chip in the future as pointed out in the following quote.

> "With multi-core its like we are throwing this Hail Mary pass down the field and now we have to run down there as fast as we can to see if we can catch it." – David Patterson

As servers are getting more powerful, each server is often set to process multiple jobs simultaneously to exploit its resources to the full capacity. This trend is observed in various forms. Virtualization technologies such as VMware products and Xen allow each individual physical server/machine to be shared by multiple virtual machines. Virtualization helps reduce the cost of maintenance, operation and provision [1], [2], and hence has been adopted in many places. Cloud computing platforms such as Amazon EC2 [3] and Microsoft Azure [4] aim at providing ubiquitous access to shared resources where resources are shared by multiple jobs and clients. In MapReduce [5] (or its open source implementation, Hadoop), which is now the de facto large data processing framework, typically several tasks are processed on the same server simultaneously. See [6]–[8] for characterizations of such tasks in Google clusters.

All the above settings can be naturally modelled as follows. Each job j has a certain computing requirement, d_j, which we call j's demand, and can be processed on a server/machine with other jobs if the total demand of the jobs does not exceed the machine's computing capacity. This model was introduced in [9]. One important assumption made in [9] was that jobs can be preempted with no penalties and no delay. However, significant overhead could occur when preempting jobs being processed to schedule other jobs. Preemption can be very costly due to context switching overheads, or may not be allowed due to system restrictions or contracts with clients. Unfortunately, non-preemption makes scheduling decisions significantly more challenging, and previous work has been focused on preemptive scheduling except for some special cases. In reality only a subset of jobs may have to be processed non-preemptively. However, in this paper we focus on developing algorithms that schedule all jobs non-preemptively. The more general setting where preemption is not allowed or limited for a subset of jobs will be studied in future work.

In this paper we study *non-preemptive* scheduling algorithms in the presence of *multiple identical machines* where multiple jobs can be scheduled simultaneously subject to the capacity constraints of individual machines. While there are various scheduling objectives considered in practice and in the literature, we consider the popular objective of minimizing the total completion time of jobs. We assume that all jobs are available to schedule from the beginning and all jobs information is known in order to focus on the offline scheduling environment where jobs have varying demands. This problem is NP-hard, and we aim at developing heuristics with *approximation guarantees*. An algorithm is said to be a c-approximation if its objective is at most c times the optimal scheduler's objective for *all* instances. We note that the non-

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

preemptive requirement makes the problem challenging as in preemptive case we can easily get a constant approximation using linear programming. To our best knowledge, there has been no theoretical study of this problem despite its wide appearance in practice.

A. Formal Problem Definition

There are N jobs, $1, 2, 3, ..., N$ that are to be scheduled on M servers/machines, which are indexed by $1, 2, 3, ..., M$. Each job j has processing time/size p_j and demand d_j. It is assumed w.l.o.g. that $p_j \geq 1$. We call the quantity $v_j := p_j d_j$ as j's volume. We will be interested in non-preemptive scheduling, meaning that a job must be processed without interruption until completion once it gets started. Each job must be assigned to a machine. A feasible schedule can be described by $\sigma = \{(S_j^\sigma, m_j^\sigma)\}_{j \in [N]}$ where S_j^σ is j's start time and m_j^σ is the machine to which j is assigned. When the schedule σ is clear from the context, we may drop σ from the superscript and simply use S_j and m_j. Note that each job j's completion time $C_j = S_j + p_j$ is determined by its start time in the non-preemptive setting. We say that a schedule σ is feasible if the total demand of jobs processed on each machine is at most the machine's capacity at all times, i.e. $\sum_{j: t \in (S_j, S_j + p_j), m_j = i} d_j \leq D_i$ for all machines i and times t. Here D_i denotes machine i's capacity. Assuming that all machines are identical, by simple scaling, we can assume w.l.o.g. that $D_i = 1$ for all machines i and $0 < d_j \leq 1$ for all jobs j. The goal is to find a feasible schedule that minimizes total completion time of all jobs, i.e. $\sum_{j \in [N]} C_j$.

B. Our Results

1) Priority-based Algorithms and Extensions to Multiple Machines: Priority-based algorithms refer to those that prioritize jobs based on fixed quantities. Such algorithms are desirable in practice in that priorities remain the same between jobs and are easy to implement. In the following priority-based algorithms, jobs are ordered in non-decreasing order of the following respective quantities.

- Shortest Job First (SJF): size p_j.
- Smallest Demand First (SDF): demand d_j.
- Smallest Volume First (SVF): volume $d_j p_j$.

In the standard/uncapacitated scheduling setting where a machine can schedule at most one job at a time, a complete ordering of jobs decides the entire schedule for the case of single machine, i.e, $M = 1$: the highest-priority unschedulued job is started at the earliest time when the machine becomes available. We can naturally extend priority-based algorithms to the capacitated setting where a machine can process multiple jobs simultaneously by starting the highest-priority unscheduled job j at the earliest time t when the job gets enough resources for p_j time steps so that it can be processed until its completion without interruption.

Each priority-based algorithm can be coupled with a machine assignment rule which decides the machine each job is scheduled on. In this work, we will consider three machine assignment rules.

- Earliest Feasible Machine (EF): Each job is assigned to the earliest feasible machine.
- Lowest Workload First (LW): Each job is assigned to the machine with the lowest workload.
- Random (RANDOM): Each job is assigned to a random machine.

We will refer to each combination as the pair of the two algorithms' names. For example, SVF-EF is SVF coupled with EF. We begin by showing that simple priority-based algorithms can have very poor performance for some instances.

Theorem 1. [Appendix A] *For any constant $c > 1$, none of the algorithms SVF, SJF, SDF is a c-approximation for minimizing total completion time. Further, this is the case even when there is only a single machine.*

The lower bound instances that fail the priority-based algorithms are simple but give a useful insight on the problem. Let's first discuss SJF. It is folklore that SJF is optimal when all jobs have a unit demand [10]. A drawback of SJF is that it could schedule slightly shorter jobs with large demands pushing back other jobs with tiny demands. The algorithm SDF can perform poorly for similar reasons. It vmay een schedule jobs in decreasing order of their sizes even when jobs have similar demands. Hence prioritizing jobs based on either their sizes or demands can lead to schedules of very poor qualities.

On the other hand, it is not very obvious at first sight why SVF is bad since it looks like a natural generalization of SJF to the capacitated setting. Our lower bound instance shows that the bad event can occur when some jobs with demands close to 1 block machines for long thus delaying all jobs with very tiny demands. However, in practice not many jobs have demands very close to machines capacities, and it is imaginable that SVF could perform well for most instances. We formalize this intuition in the following theorem.

Theorem 2. [Section II] *For a constant $0 < \alpha < 1$, if all jobs have demands at most α, then SVF-EF is a $\left(\frac{3\alpha}{1-\alpha} + 3\right)$-approximation. More precisely, SVF-EF's total completion time is at most $\frac{1}{1-\alpha}$ times the optimum plus $\frac{2}{1-\alpha}$ times the total size of all jobs.*

When the system is overloaded, the contribution of jobs sizes to the total completion time objective becomes negligible. Hence, Theorem 2 implies that SVF-EF's objective becomes arbitrarily close to the optimum as α tends to 0 and the system gets more overloaded. We note that SVF was analyzed only together with EF since it does not seem to yield similar approximation guarantees when coupled with other machine assignment rules.

2) Constant Approximations: Finally, building on the intuition, along with the above results, we develop algorithms that are constant approximations for *all* instances for the first time.

Theorem 3. [Section III] *For any constant $\epsilon > 0$, there is a $12(1+\epsilon)$-approximation when $M = 1$. When there are more than one machine, there is a $5 + O(1/M)$-approximation.*

When there are multiple machines ($M \geq 2$), we combine SJF and SVF. Let us call jobs with demands higher than 1/2 as high-demand jobs, and others as low-demand jobs. It is easy to see that no two high-demand jobs can be processed on the machine at the same time. Intuitively, scheduling high-demand jobs in the capacitated setting is similar to scheduling jobs in the non-capacitated setting. Hence we dedicate some machines to processing high-demand jobs separately, and process low-demand jobs on the remaining machines. Thus, by separating high-demand jobs from low-demand jobs, we achieve a constant approximation.

However, when there is only one machine ($M = 1$), we need a different idea. In this case, we reduce the problem to the geometric packing problem where the goal is to maximize the total profit of rectangles packed that can be packed into a target rectangle [11]. Using this reduction and borrowing ideas from [12], for any time t, we can obtain a partial schedule that completes as many jobs by time $3(1+\epsilon)t$ as the optimal scheduler does by time t. Then, we derive a complete schedule by concatenating the partial schedules for geometrically increasing t. We note that it is possible to reduce this problem to the maximum throughput scheduling problems [13], [14]. However, the resulting approximation ratio is worse, hence we use the above reduction.

As discussed earlier, any reasonably good algorithms seem to have to take into account both quantities, jobs sizes and demands. Then, a natural question is if we really need a delicate 2D-packing to obtain constant approximations. Quite surprisingly, we show that this is not the case. In particular, we show one can achieve a constant approximation in a nearly linear time. While the approximation guarantee is worse than the previous algorithms, we include this result since it shows the possibility of existence of more efficient algorithms with approximation guarantees comparable to those claimed in Theorem 3. Due to the space of constraints, the proof of the following theorem is deferred to the full version of this paper.

Theorem 4. [Section IV] *There is a constant approximation where all jobs processed at the same time have an equal size within a factor of 2. Further, the algorithm runs in $O(N \log N)$ time.*

This algorithm is based on reduction to bin packing. The key idea is to group jobs of similar sizes so that each 'consolidated' job has a sufficiently large demand. Then, we schedule the consolidated blocks mimicking the algorithm Highest Density First (HDF). The algorithm HDF is a generalization of SJF to the case where jobs have weights, and is known to be constant-approximate in the non-capacitated setting. The actual algorithm needs minor modifications, but this is a high-level description of our algorithm and the intuition. While the high-level idea is intuitive, the analysis is non-trivial.

3) Experiment Results: We conduct experiments via simulation on both synthetic and real world data sets to support our theoretical findings (Section V). We implemented priority-based algorithms combined with three different machine assignment rules as explained in the beginning of this section. Experiments show that SVF outperforms all other priority-based algorithms when combined with the EF assignment rule that assigns the highest-priority unscheduled job to the earliest available machine. Initial experiments show that grouping jobs of similar size beats SVF-EF when the number of jobs in the workload are large enough. However, we present the complete details of experiments as well as analysis in the full version of the paper. We believe our work is of fundamental importance since it not only develops the first constant approximations for the non-preemptive capacitated scheduling problem commonly arising in many scheduling environments, but also provides in-depth insights on the problem. Further, we view the natural greedy algorithm SVF through the lens of approximations, and give an explanation on why SVF distinguishes itself among other algorithms.

C. Related Work

Due to the space constraints, we only discuss the most related work. When each job has a release time and deadline, and the goal is to maximize the total weight of jobs completed before their deadline, several approximations are known [13], [14]. If the objective is minimizing the makespan, we can easily adapt our algorithms and analysis to obtain constant approximations (Section III).

As mentioned, [9] studied our problem in the preemptive setting, and proposed and analyzed several online algorithms for the total flow time objective under the resource augmentation model; a job j's flow time is defined as the job's completion time minus its arrival time. In contrast, our work studies non-preemptive scheduling. In the non-preemptive setting, the flow time objective becomes very difficult to approximate [15]. Hence, we consider the completion time objective which admits more positive results. For works in the queueing setting where jobs arrive following certain distributions, see [16], [17]. In such models, typically the focus is on the stability of the system, which is very different from our paper.

A lot of work was done in somewhat related parallelization models. For example, see [18]–[22]. Roughly speaking, in these works, one can schedule a job on multiple machines/processors simultaneously to speed up the processing. While these models accurately capture how jobs are parallelized at a high level, they do not enforce hard constraints on jobs demands. On the other hand, in our model, a job cannot be processed when given less resources than it demands.

II. Analysis of Smallest Volume First for Jobs with Small Demands

In this section we prove Theorem 2. Since SVF will be paired only with EF, we may refer to SVF-EF simply as SVF. Similarly, another algorithm SJF, which is considered for the sake of analysis, will be paired with EF, and we refer to SJF-EF simply as SJF. It is well known that SJF is optimal in the non-capacitated case. We first show that the optimal schedule can only be better off if it can compress jobs: replace each job j with a job j' with demand 1 and size $p_j d_j$ preserving

the volume of the job. Let I be the original instance and I' the compressed instance. For notational convenience, for an instance I'', we allow $\text{OPT}(I'')$ to denote a fixed optimal schedule for instance I'' or its objective depending on the context.

Lemma 1. $\text{OPT}(I') \leq \text{OPT}(I)$.

Proof: Consider a fixed machine m. Let $j_1, j_2, ..., j_k$ be the jobs assigned to the machine m, which are ordered in their completion times in schedule $\text{OPT}(I)$. Let $j'_1, j'_2, ..., j'_k$ be the compressed jobs corresponding to $j_1, j_2, ..., j_k$. We process jobs $j'_1, j'_2, ..., j'_k$ in this order on the same machine. It is easy to see that we can complete each job j'_κ before j_κ completes in $\text{OPT}(I)$. This is because no schedule can complete all jobs $j_1, j_2, ..., j_\kappa$ before time $\sum_{h=1}^{\kappa} p_h d_h$ which is the total volume of jobs $j_1, ..., j_\kappa$, and compression preserves each job's volume. This completes the proof. ∎

Further, we know that SJF is optimal for the instance I' since I' is an instance in the non-capacitated setting [23]. Hence we can assume w.l.o.g. that the optimal schedule $\text{OPT}(I')$ for I' is generated by SJF. Let S_j and C_j denote job j's start and completion times in the former schedule, respectively. Define S^*_j and C^*_j analogously for the latter schedule $\text{OPT}(I')$. The reader may wonder if compression gives too much power to the optimal scheduler. For example, it may finish a long job with a tiny demand very quickly by compressing it. Then, $C^*_j - S^*_j$ could be much smaller than p_j. Hence we will bound C_j by S^*_j and p_j in Theorem 2. We now show that SVF utilizes resources pretty well in the following sense.

Lemma 2. *Suppose SVF starts processing job j at time S_j. Then it must be the case that each machine's capacity is used by at least $1 - \alpha$ from time 0 to S_j.*

Proof: To prove the lemma we show the following invariant. Fix a machine m. For notational convenience, let $1, 2, ..., N$ be the order of jobs considered by SVF. Let $D_{j,t}$ denote the total demand of jobs amongst $1, 2, 3, ..., j$ that are processed at time t on machine m. We claim that for any fixed j, $\max\{D_{j,t}, 1 - \alpha\}$ is non-increasing in time t. We prove this by induction on j. The claim trivially holds when $j = 1$. Assume that the claim is true for $j - 1$. Note that $D_{j-1,t} > 1 - \alpha$ for all $0 \leq t < S_j$. Otherwise, knowing that $D_{j-1,t}$ is non-increasing in t, we could have started processing job j earlier than time S_j. If job j gets processed on machine m, we have $D_{j,t} = D_{j-1,t} + d_j$ for all $S_j < t < S_j + p_j$. Since $D_{j,t} > 1 - \alpha$ for $0 \leq t < S_j$, $D_{j-1,t} \leq 1 - \alpha$ for $t > S_j$, and $D_{j-1,t}$ is non-increasing in t for times greater than S_j, the claim also holds true for job j. If job j gets processed on other machines, we have $D_{j,t} = D_{j-1,t}$, which immediately implies the claim for job j due to the induction hypothesis. ∎

Next, we compare each job's start time in both schedules, those by SVF for I and by SJF for I'. We first upper bound each job's start time in SVF's schedule.

Lemma 3. *For all jobs j, we have $S_j \leq \frac{1}{(1-\alpha)M} \sum_{h=1}^{j-1} v_h$.*

Proof: By Lemma 2, we know that SVF processed at least $(1 - \alpha)$ volume of work for jobs $1, 2, ..., j - 1$ by time S_j. Hence, we have $\sum_{h=1}^{j-1} v_h \geq (1 - \alpha) M S_j$. ∎

Lemma 4. *For all jobs j, we have $S^*_j \geq \frac{1}{M} \sum_{h=1}^{j-1-(M-1)} v_h$.*

Proof: In SJF's schedule, at most $M - 1$ jobs are being processed when job j starts getting processed. Hence SJF must complete at least $j - 1 - (M - 1)$ jobs out of $1, 2, ..., j - 1$ by time S^*_j. Thus, SJF processes at least $\sum_{h=1}^{j-1-(M-1)} v_h$ volume of work by time S^*_j, which yields the lemma. ∎

We are now ready to complete the proof of Theorem 2. For each job j we derive,

$$\begin{aligned}
C_j &\leq p_j + \frac{1}{(1-\alpha)M} \sum_{h=1}^{j-1} v_h \quad \text{[By Lemma 3]} \\
&= p_j + \frac{1}{(1-\alpha)M} \Big(\sum_{h=1}^{j-1-(M-1)} v_h + \sum_{h=j-M+1}^{j-1} v_h \Big) \\
&\leq p_j + \frac{1}{(1-\alpha)M} \Big(\sum_{h=1}^{j-1-(M-1)} v_h + \sum_{h=j-M+1}^{j-1} v_j \Big) \\
&\leq p_j + \frac{1}{1-\alpha} \cdot S^*_j + \frac{M-1}{M(1-\alpha)} v_j \quad \text{[By Lemma 4]} \\
&\leq \frac{1}{1-\alpha} \cdot S^*_j + \frac{2}{1-\alpha} \cdot p_j
\end{aligned}$$

The last inequality follows since job j has demand at most α, meaning $v_j \leq \alpha p_j$. Knowing that $\sum_j S^*_j$ and $\sum_j p_j$ is at most equal to the optimal total completion time, we have Theorem 2.

III. Constant Approximations for All Instances

In the previous section we showed that SVF is a constant approximation if all jobs have demands at most α that is a constant smaller than 1. In this section, we develop algorithms that are $O(1)$-approximate for all instances, proving Theorem 3. We consider two cases depending on whether $M \geq 2$ or not.

A. Single Machine Case ($M = 1$)

In this case, we reduce our problem to the 2D-strip packing problem. Using this reduction, we will show the following.

Lemma 5. *Suppose there is a subset of n jobs that can be completed within L time steps. Then, for any constant $\epsilon > 0$, one can find a schedule in polynomial time that completes at least n jobs by time $3(1 + \epsilon)L$.*

What this lemma means is that we can complete as many jobs as the optimal scheduler if we are allowed to use $3(1+\epsilon)$ factor more time steps. Before proving the lemma, We first discuss how we use it to prove Theorem 3 in the single machine case. By repeatedly using this lemma, we obtain partial schedules and concatenate them to get a final schedule. Let N_ℓ denote the total number of jobs that the optimal schedule completes by time 2^ℓ. Using Lemma 5, we find a set of at least N_ℓ jobs that can be scheduled by time

$3(1+\epsilon)2^\ell$ – let this schedule be denoted by B_ℓ, which we call a block. We concatenate the blocks $B_0, B_1, B_2, ...$ in this order. If a job in B_ℓ was already scheduled in the previous block, we simply remove the job from the block B_ℓ. The resulting schedule is clearly feasible. Note that jobs in block B_ℓ are processed between times $3(1+\epsilon)\sum_{h=0}^{\ell-1} 2^h$ and $3(1+\epsilon)\sum_{h=0}^{\ell} 2^h = 3(1+\epsilon)\cdot(2^{\ell+1}-1)$. For notational simplicity, let $N_{-1} := 0$. In this schedule, the total completion time is at most

$$3(1+\epsilon)\sum_{\ell \geq 0} 2^\ell (N - N_{\ell-1}) \tag{1}$$

This is because when the block B_ℓ is scheduled, there are at most $N - N_{\ell-1}$ jobs alive, and the block is scheduled for $3(1+\epsilon)2^\ell$ time steps. On the other hand, the optimal total completion time is at least

$$N + \sum_{\ell \geq 1} 2^{\ell-1}(N - N_\ell), \tag{2}$$

since the optimal schedule has at least $N - N_\ell$ jobs alive during the time interval $(2^{\ell-1}, 2^\ell)$ for $\ell \geq 1$. The first term N follows since no job completes before time 1; recall that all jobs have sizes at least 1. A simple algebra gives Theorem 3 for the case $M = 1$.

It now remains to prove Lemma 5.

*Proof of L*emma 5: We borrow ideas from [11] which studies the problem of maximizing the total profit of rectangles that can be packed into a target rectangle without overlap; here rectangles can only be moved vertically and horizontally, and rotations are not allowed. We first find a set of jobs J whose total volume does not exceed $(1+\epsilon)L$. This is essentially a special case of Knapsack problem where we are asked to pack items of different profits and sizes into a knapsack with the goal of maximizing the total profit of items packed. Then, it is well known that we can pack in polynomial time as many items into a knapsack of size $(1+\epsilon)L$ as the optimal solution can pack into a knapsack of size L; for example see [24].

We now view the scheduling instance as an input to the 2D-strip packing problem. In the 2D-strip packing problem, we are asked to pack all given rectangles without rotations into a strip with bounded width but with unbounded height, and the goal is to minimize the strip height. Towards this end, for each job j, create a rectangle $r(j)$ with width p_j and height d_j. Steinberg [12] shows a sufficient condition that all rectangles can be packed – particularly, the condition is satisfied if the strip height is at least twice the maximum height of rectangles, and the total area of rectangles is at most half of the strip area (and with other 'easy conditions'). Hence we can pack all rectangles corresponding to J into a strip with width $(1+\epsilon)L$ and height 2. Following ideas in Section 3 of [11], we can pack all the rectangles corresponding to J into three strips with width $(1+\epsilon)L$ and height 1. We obtain the desired schedule by concatenating these three strips horizontally and translating it into a schedule. □

B. Multiple Machines Case ($M \geq 2$)

We schedule jobs of demands more than 1/2 (high-demand jobs) and the other jobs (low-demand jobs) separately on two disjoint sets of machines, $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. Our algorithm, which we call HYBRID, combines two algorithms, SJF and SVF. We use SJF to schedule high-demand jobs on $\mathcal{M}_1$ pretending that high-demand jobs have demands equal to 1. For low-demand jobs, we schedule them on $\mathcal{M}_2$ using SVF. As in Section II, we use S_j and C_j to denote job j's starting and completions times in the schedule of our algorithm.

1) Low-demand Jobs: We first upper bound low-demand jobs' contribution to the objective. Obviously, the optimal scheduler can only decrease its objective if it only needs to complete low-demand jobs. Hence we can assume w.l.o.g. that we only have low-demand jobs. The analysis is very similar to that in the previous section. The only difference is that SVF can only use M_2 machines while the optimal scheduler can use all M machines. As before, let S_j^* and C_j^* denote the start and completion times of a low-demand job j in the optimal schedule respectively, assuming that all jobs are compressed. Then, we can easily adapt Lemmas 3 and 4 as follows.

Lemma 6. *For all low-demand jobs j, we have*

- $S_j \leq \frac{2}{M_2}\sum_{h=1}^{j-1} v_h$.
- *For all jobs j, we have $S_j^* \geq \frac{1}{M}\sum_{h=1}^{j-1-(M-1)} v_h$.*

Lemma 7. *For all low demand jobs j, we have* $C_j \leq \frac{2M}{M_2}\cdot S_j^* + \left(\frac{M-1}{M_2}+1\right)\cdot p_j$

Proof:

$$\begin{aligned}
C_j &\leq p_j + \frac{2}{M_2}\sum_{h=1}^{j-1} v_h \quad \text{[Lemma 6]}\\
&= p_j + \frac{2}{M_2}\left(\sum_{h=1}^{j-1-(m-1)} v_h + \sum_{h=j-m+1}^{j-1} v_h\right)\\
&\leq p_j + \frac{2}{M_2}\left(\sum_{h=1}^{j-1-(m-1)} v_h + \sum_{h=j-m+1}^{j-1} v_j\right)\\
&\leq p_j + \frac{2M}{M_2}\cdot S_j^* + \frac{2(M-1)}{M_2} v_j \quad \text{[Lemma 6]}\\
&\leq p_j + \frac{2M}{M_2}\cdot S_j^* + \frac{M-1}{M_2} p_j \quad \text{[Since } v_j \leq \tfrac{1}{2}p_j]\\
&\leq \frac{2M}{M_2}\cdot S_j^* + \left(\frac{M-1}{M_2}+1\right)\cdot p_j
\end{aligned}$$

■

2) High-demand Jobs: We now shift our attention to high-demand jobs. We can assume w.l.o.g. that the optimal scheduler only needs to complete high-demand jobs on the M machines. For notational convenience, assume that high-demand jobs $1, 2, 3, ...$ are ordered in non-decreasing order of their sizes. Knowing that at most one high-demand job can be processed at any time, we can also assume w.l.o.g. that optimal schedule is produced by SJF.

Lemma 8. *For all high-demand jobs j, we have*

- $S_j \leq \frac{1}{M_1} \sum_{h=1}^{j-1} p_h$.
- *For all jobs j, we have* $S_j^* \geq \frac{1}{M} \sum_{h=1}^{j-1-(M-1)} p_h$.

Proof: The upper bound on S_j follows from the fact that at most one machine gets available for scheduling job j before time $\frac{1}{M_1} \sum_{h=1}^{j-1} p_h$. The lower bound on S_j^* follows since SJF completes all jobs shorter than j but possibly other jobs being processed on the other $M-1$ machines at time S_j^*. ■

Lemma 9. *For all high demand jobs j, $C_j \leq (\frac{M-1}{M_1}+1) \cdot C_j^*$*

Proof:

$$\begin{aligned} C_j &\leq p_j + \frac{1}{M_1} \sum_{h=1}^{j-1} p_h \quad \text{[Lemma 8]} \\ &= p_j + \frac{1}{M_1} \left(\sum_{h=1}^{j-1-(M-1)} p_h + \sum_{h=j-M+1}^{j-1} p_h \right) \\ &\leq p_j + \frac{M}{M_1} \cdot S_j^* + \frac{M-1}{M_1} p_j \quad \text{[Lemma 8]} \\ &\leq (\frac{M-1}{M_1} + 1) \cdot C_j^* \end{aligned}$$

In the last inequality we used the fact that $C_j^* = S_j^* + p_j$. ■

3) Putting Pieces Together: By summing the inequalities in Lemma 7 over all low-demand jobs, we have that HYBRID is a $(\frac{3M-1}{M_2}+1)$-approximation for low-demand jobs. By summing over the inequalities in Lemma 9 over all high-demand jobs, we have that HYBRID is a $(\frac{M-1}{M_1}+1)$-approximation for high-demand jobs. We set $M_1 = \lfloor \frac{M-2}{4} \rfloor + 1$ and $M_2 = \lfloor \frac{3(M-2)}{4} \rfloor + 1$. A simple algebra gives Theorem 3 for the multiple machines case.

IV. Constant Approximations by Grouping Jobs of Similar Sizes

The constant approximations we gave in the previous section try to pack jobs efficiently into machines. In particular, in the single machine case, the algorithm uses a delicate geometric packing. In this section, we show that even if we only process jobs of similar sizes simultaneously, we can still obtain constant approximations. Before we describe our algorithm, we do the following simple preprocessing. The loss in the approximation ratio will be factored in at the end of analysis. Due to the space constraints, most of the proofs are deferred to the full version of this paper.

Proposition 1. *We can assume w.l.o.g. that each job size is a power of two with a loss of factor two in the approximation ratio.*

From now on, we assume that each job size is a power of two. We will say that job j is in class k and denote it as $j \in G_k$ if its size is 2^k. Our algorithm consists of three main steps, packing jobs into blocks, ordering blocks, and assigning blocks to machines. The first step uses the well-known algorithm, First-Fit for the Bin Packing problem; for the definition of the problem and related results, see [24].

Block-Scheduling Algorithm:

1) Packing jobs into blocks: For each class k, create blocks using the First-Fit algorithm: Consider jobs in class k in non-decreasing order of their demands, and partition them into groups so that the total demand of jobs in every group does not exceed 1. Here, we create another group only when the currently considered job cannot fit with other jobs into the existing groups. We call the created group as blocks, and let $B_{k,l}$ denote lth block we created for jobs in class k.
2) Ordering blocks: Let p_B denote the size of block B, which is defined as the size of any job packed into the block. Let N_B as the number of jobs in B. Define B's density, $\eta_B := N_B / p_B$. Blocks are ordered in non-increasing order of their densities.
3) Assigning blocks to machines: We will recursively schedule blocks as follows. In the kth step, for each machine m from 1 to M we find a maximal set of unscheduled blocks $\mathcal{B}_k^{A,m}$ of sizes at most 2^k with the highest densities such that the total size of blocks in $\mathcal{B}_k^{A,m}$ does not exceed $\gamma \cdot 2^k$ on each machine, where $\gamma := (9 + \lceil 6M \rceil)$. We order blocks assigned in $\mathcal{B}_k^{A,m}$ in non-increasing order of their densities.

We focus on proving the approximation guarantee since other claims in Theorem 4 immediately follow from the algorithm's description. As in Section II, for the sake of analysis, we will assume w.l.o.g. that the optimal scheduler is allowed to compress jobs; compression can only help the optimal scheduler. Recall that for each job j, its compressed version j' has demand 1 and size $p_j d_j$. However, here the optimal scheduler can use compression in a more restrictive way. Note that the total completion time of jobs is equal to the sum of the total number of alive jobs over time. For *each* time t, the optimal scheduler tries to minimize the number of jobs alive at time t. That is, the optimal scheduler's only goal is to try to complete as many jobs as possible by time t. While the optimal scheduler can compress jobs, it is *not* allowed to finish any job with sizes more than t. We can assume w.l.o.g. that the optimal scheduler completes those with smaller demands first amongst jobs of an equal size. Let R_t^* denote the number of jobs this strengthened optimal scheduler (denoted as OPT) has left at time t. The careful reader may notice that such schedules may not be consistent across different times t. However, $\int_{t \geq 0} R_t^*$ clearly lower bounds the optimal total completion time. We let R_t denote the number of jobs alive at time t in the schedule of our algorithm, which we will refer to as A.

We show the following key lemma, which will complete the proof of Theorem 4.

Lemma 10. *For all times $t \geq 0$, $R_{2\gamma t} \leq R_t^*$.*

Proof of [Theorem 4]

$$\int_{t=0}^{\infty} R_t \; dt = 2\gamma \int_{t=0}^{\infty} R_{2\gamma t} \; dt \leq \int_{t=0}^{\infty} R_t^* \; dt.$$

Knowing that the first [last, resp.] integral is A's [OPT's, resp.] total completion time, and factoring in the approximation

loss stated in Proposition 1, we derive that A ia a 4γ-approximation. □

V. Experimental results

A. Algorithms Implemented

We use simulation experiments to compare the performance of priority-based algorithms. In this work, we focused on priority-based algorithms since they seem more practical due to the simplicity. As mentioned earlier, each priority based algorithm can be combined with any machine assignment rule. We consider the following category of algorithms that have two main scheduling components:

1) Priority rule: This rule orders jobs based on certain quantities. Jobs will be dispatched to machines in this order.
2) Machine assignment rule: This rule decides to which machine to dispatch each job in the ordered list.

We have several choices for priority rule as discussed in Section I-B: Shortest Job First (SJF), Smallest Demand First (SDF) and Smallest Volume First (SVF). Our experiments showed that SDF had poor performance in comparison with SJF and SVF so we discarded the related results to improve the readability of performance comparison plots. For the machine assignment rule, we considered two algorithms. One is assigning the currently considered job to the machine with the lowest workload (LW). The other is choosing the machine where the job's completion time is minimized (EF). In our experiments we also considered random ordering of jobs and random machine assignment for purpose of comparison. Hence by coupling the three choices for the priority rule (RANDOM, SJF, SVF) with the three choices for the machine assignment rule (RANDOM, LW, EF), we compare 9 different algorithms in our experiments. We call each algorithm by combining the names of the paired rules.

B. Simulation Methodology

We developed a discrete-event simulator in Java which consists of parallel machines with bounded capacities. In all experiments presented here except the last experiment, the system model consists of 50 identical machines with resource capacity of 200 CPUs on each. Each job is described by job size and a number for CPU requirement. We assume that preemption is not allowed and all jobs are available at the beginning. For simplicity, we assume that each job consists of only one task.

C. Workloads

We assume jobs' CPU demands are distributed uniformly in interval $[1, 100)$ with probability 0.75 and in interval $[100, 200]$ with probability 0.25. For job sizes we assumed two distributions:

1) SYNTH1: Distributed uniformly in interval $[1, 100]$ with probability 0.7, in interval $[300, 350]$ with probability 0.15, and in interval $[450, 500]$ with probability 0.15 .
2) SYNTH2: Distributed geometrical with mean equal to that of the uniform distribution for SYNTH1.

We also performed experiments with a real world workload dataset from an online benchmark repository [25]. We chose the HPC2N dataset which includes job sizes and CPU requirements. We ignored details like job submission times included in SWF format [26] as we assume all jobs are available for schedule at the beginning. We run our algorithms on a sample of 800 jobs randomly chosen from HPC2N.

D. Experiments

In this section we evaluate the performance of priority based algorithms on synthetic and real world datasets.

In the first set of experiments we compared the performance of the algorithms as a function of workload size, i.e, the number of jobs. We generated Workload sizes from 3000 to 16000 with increments of 1000s from SYNTH1 and SYNTH2. For each workload size we generated data 10 times and computed the average. Figure 1 plots average completion time vs. number of jobs using the synthetic workload.

All experiments were conducted by running synthetic workloads on 50 machines. As plots of SJF-RANDOM, SVF-RANDOM and SVF-LW in figure 1.a and plots of SJF-RANDOM, SJF-LW, SVF-RANDOM and SVF-LW in figure 1.b are overlapping, we zoomed into the graph for better illustration. The relative performance of different priority rules are similar in both distributions. However, the plots for SYNTH2 are more distinct and SVF-EF diverges from other adjacent curves more rapidly as can be seen in figure 1.b. Compatible with our theoretical findings (Theorems 1 and 3), SVF-EF demonstrates the best performance, considerably outperforming SJF-EF. Further, SVF-EF's superiority in performance becomes more distinguished as the workload size increases.

In second experiment we tested our algorithms on a sample of 800 jobs randomly selected from a real-world trace HPC2N. The comparison of average completion times is shown in Figure 2. It can be seen in the figure that SVF-EF has the best performance for the real-world data sets.

Fairness is another important scheduling criteria in high performance computing systems. As we are minimizing average completion time, it could happen that larger jobs get more benefits than smaller jobs as they contribute more to the objective function. A common measure for fairness in literature is *stretch* which measures by what factor, a job is slowed down relative to the time it takes on an underloaded system [27]–[29].

In the last experiment, we compare the average, maximum and standard deviation of the different algorithms for 5000 jobs generated from SYNTH1 on 10 machines as shown in table I. For the random ordering, we see a large range of stretch values which is reflected in the maximum and standard deviation values. It is observable from the table that SJF-EF and SVF-EF have the best average and maximum stretches. The SJF-EF performs slightly better than SVF-EF, as sorting jobs in their order of increasing sizes is more likely to finish shorter jobs earlier, which are more sensitive to delays. The algorithm SVF-EF is the most preferable since it outperforms

other algorithms in average completion time while achieving a pretty good fairness.

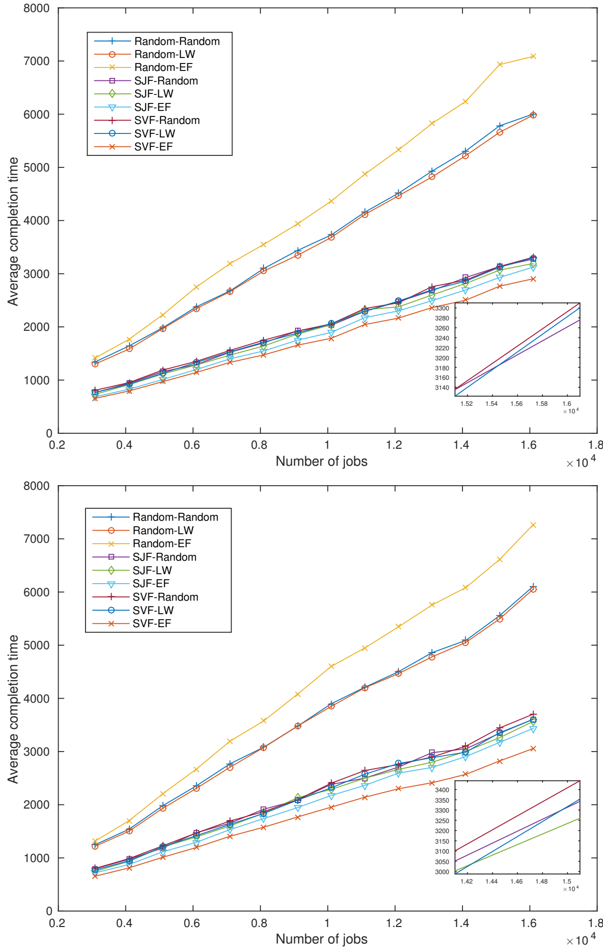

Fig. 1: Average completion time vs. number of jobs for synthetic data set, The different trend for increase in average total completion time is observable as number of jobs are increased from 3000 to 16000 a. Job sizes are generated from uniform distribution. b. Job sizes are generated from geometric distribution.

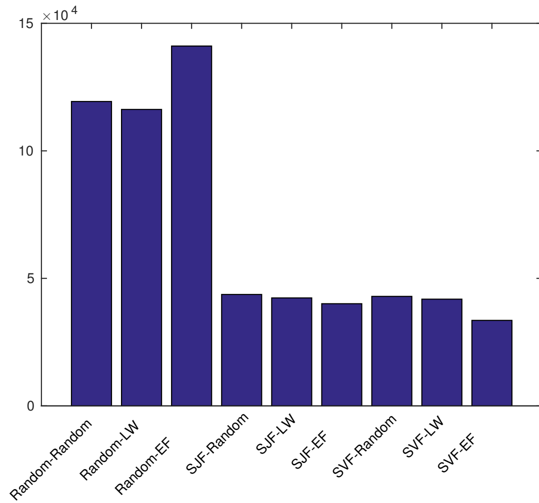

Fig. 2: Average completion time comparison for 800 jobs sample from HPC2N dataset.

Method	Average	Max	std
RANDOM-RANDOM	507.80	22011	1385.19
RANDOM-LW	503.00	21516	1371.94
RANDOM-EF	514.13	19056	1265.38
SJF-RANDOM	64.85	835	72.72
SJF-LW	61.03	1470	85.44
SJF-EF	**39.82**	99.86	25.49
SVF-RANDOM	83.91	2235	142.25
SVF-LW	80.83	1977	129.25
SVF-EF	**40.81**	131.08	33.63

TABLE I: Comparison of stretch in the algorithms, 5000 jobs are run on 10 parallel machines

VI. Concluding Remarks

In this paper we proposed new algorithms for scheduling jobs with varying demands on multiple machines without interruption. For this scheduling scenario which is widely observed in practice, we considered one of the most popular objectives, minimizing total completion time. Our work gives in-depth insights on the problem, and develops heuristics with provable approximation guarantees. In particular, we analyzed a simple thus scalable algorithm, Smallest Volume First (SVF) and showed why it outperforms other algorithms, which is verified by our experiments. We also gave constant approximations for the first time. We believe that our work is only a starting point for many interesting future directions. A more general assumption can be considered by taking into account different types of resources. Such heterogeneous resources can be modelled by demand and supply vectors over multiple resources such as CPU and memory. Further, machines may have different amounts of resources. Also it would be interesting to consider the case where each job has a demand that changes over time.

Acknowledgements

S. Im was supported in part by NSF Award CCF-1008065.

References

[1] "http://www.vmware.com/consolidation/overview/."

[2] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," in *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*. IEEE, 2007, pp. 119–128.

[3] "http://aws.amazon.com/ec2/."

[4] "http://www.windowsazure.com/."

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[7] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in google compute clusters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 3.

[8] Q. Zhang, J. L. Hellerstein, and R. Boutaba, "Characterizing task usage shapes in googles compute clusters," in *Large Scale Distributed Systems and Middleware Workshop (LADIS11)*, 2011.

[9] K. Fox and M. Korupolu, “Weighted flowtime on capacitated machines,” in *ACM-SIAM SODA*. SIAM, 2013, pp. 129–143.

[10] W. E. Smith, “Various optimizers for single-stage production,” *Naval Research Logistics Quarterly*, vol. 3, no. 1-2, pp. 59–66, 1956.

[11] K. Jansen and G. Zhang, “Maximizing the total profit of rectangles packed into a rectangle,” *Algorithmica*, vol. 47, no. 3, pp. 323–342, 2007.

[12] A. Steinberg, “A strip-packing algorithm with absolute performance bound 2,” *SIAM Journal on Computing*, vol. 26, no. 2, pp. 401–409, 1997.

[13] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber, “A unified approach to approximating resource allocation and scheduling,” *Journal of the ACM (JACM)*, vol. 48, no. 5, pp. 1069–1090, 2001.

[14] S. Albagli-Kim, H. Shachnai, and T. Tamir, “Scheduling jobs with dwindling resource requirements in clouds,” in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 601–609.

[15] N. Bansal, H.-L. Chan, R. Khandekar, K. Pruhs, B. Schicber, and C. Stein, “Non-preemptive min-sum scheduling with resource augmentation,” in *IEEE FOCS*. IEEE, 2007, pp. 614–624.

[16] S. T. Maguluri and R. Srikant, “Scheduling jobs with unknown duration in clouds,” *Networking, IEEE/ACM Transactions on*, vol. 22, no. 6, pp. 1938–1951, 2014.

[17] S. T. Maguluri, R. Srikant, and L. Ying, “Heavy traffic optimal resource allocation algorithms for cloud computing clusters,” *Performance Evaluation*, vol. 81, pp. 20–39, 2014.

[18] J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng, “Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics,” *J. Scheduling*, vol. 6, no. 3, pp. 231–250, 2003.

[19] J. Blazewicz, M. Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz, “Preemptable malleable task scheduling problem,” *IEEE Trans. Computers*, vol. 55, no. 4, pp. 486–490, 2006.

[20] P. Sanders and J. Speck, “Energy efficient frequency scaling and scheduling for malleable tasks,” in *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, 2012, pp. 167–178.

[21] H. Chang, M. Kodialam, R. R. Kompella, T. Lakshman, M. Lee, and S. Mukherjee, “Scheduling in mapreduce-like systems for fast completion time,” in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 3074–3082.

[22] Y. Zheng, N. B. Shroff, and P. Sinha, “A new analytical technique for designing provably efficient mapreduce schedulers,” in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 1600–1608.

[23] M. L. Pinedo, *Scheduling: theory, algorithms, and systems*. Springer Science & Business Media, 2012.

[24] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge University Press, 2011. [Online]. Available: http://www.cambridge.org/de/knowledge/isbn/item5759340/?site_locale=de_DE

[25] “Parallel workloads archive,” http://cs.huji.ac.il/labs/parallel/workloads, note = accessed: 2015-07-01.

[26] D. G. Feitelson, D. Tsafrir, and D. Krakov, “Experience with using the parallel workloads archive,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967–2982, 2014.

[27] M. Stillwell, F. Vivien, and H. Casanova, “Dynamic fractional resource scheduling for hpc workloads,” in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[28] M. A. Bender, S. Muthukrishnan, and R. Rajaraman, “Approximation algorithms for average stretch scheduling,” *Journal of Scheduling*, vol. 7, no. 3, pp. 195–222, 2004.

[29] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, “Characterization of backfilling strategies for parallel job scheduling,” in *Parallel Processing Workshops, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 514–519.

Appendix

A. Lower bounds for Simple Priority Based Algorithms

This section is devoted to prove Theorem 1. We assume throughout this section that there is only a single machine. We note that the lower bounds for SJF and SDF hold even if jobs have demands less than any fixed constant.

a) Lower Bound for SJF: Consider the following instance. There are two types of jobs. Type-A jobs have demands 1/2 and sizes 1. Type-B jobs have demands $\frac{1}{2N}$ and sizes $1+\epsilon$ where $\epsilon > 0$ is an arbitrarily small parameter. There are $\sqrt{N}$ type-A jobs, and the other $N - \sqrt{N}$ jobs are type-B.

The algorithm SJF will first complete Type-A jobs by time $\sqrt{N}/2$. Since no type-B jobs get processed before time $\sqrt{N}/2$, the total completion time by SJF is at least $(N - \sqrt{N}) \cdot \sqrt{N}/2 = \Omega(N^{1.5})$. On the other hand, we can complete all Type-B jobs by time 1, and all Type-A jobs by time $(\sqrt{N})/2+1$. Hence the optimal total completion time is at most $(N - \sqrt{N}) \cdot 1 + \sqrt{N} \cdot ((\sqrt{N})/2+1) = O(N)$, which gives a gap of $\Omega(\sqrt{N})$.

b) Lower Bound for SDF: The lower bound instance is as follows. As before there are two types of jobs. The unique Type-A job has demand $(1-\epsilon)$ and size N where $\epsilon > 0$ is a parameter that is arbitrarily small. Type-B jobs have demands 1 and size $1/N$. All the N jobs are type-B except the unique Type-A job. The algorithm SDF processes no type-B jobs until time N, hence has total completion time at least $N(N-1)$. However, we can complete all type-B jobs by time 1 and the type-A job by time N+1, hence the optimal total completion time is at most $(N-1) \cdot 1 + 1 \cdot (N+1) = 2N$. Hence we obtain a gap of $\Omega(N)$.

c) Lower Bound for SVF: Again, there are two types of jobs. The unique Type-A jobs has size N and demand $\frac{1}{2N^2}$. All the other $N-1$ jobs are Type-B and they have size $1/N$ and demand 1. The algorithm SVF starts processing the type-A jobs. Note that type-B jobs cannot be processed until the job-A completes since they require the full capacity to get processed. Hence SVF has total completion time at least $N(N-1)$. In contrast, the optimal total completion time is at most $2N$ since we can complete type-B jobs by time 1 and the type-B job by time $N+1$, which implies a gap of $\Omega(N)$.