Dynamic Fractional Resource Scheduling versus Batch Scheduling

Mark Stillwell, Frédéric Vivien, Henri Casanova

Dynamic Fractional Resource Scheduling vs. Batch Scheduling

Mark Stillwell, Frédéric Vivien, and Henri Casanova

Abstract—We propose a novel job scheduling approach for homogeneous cluster computing platforms. Its key feature is the use of virtual machine technology to share *fractional* node resources in a precise and controlled manner. Other VM-based scheduling approaches have focused primarily on technical issues or extensions to existing batch scheduling systems, while we take a more aggressive approach and seek to find heuristics that maximize an objective metric correlated with job performance. We derive absolute performance bounds and develop algorithms for the online, non-clairvoyant version of our scheduling problem. We further evaluate these algorithms in simulation against both synthetic and real-world HPC workloads and compare our algorithms to standard batch scheduling approaches. We find that our approach improves over batch scheduling by orders of magnitude in terms of job stretch, while leading to comparable or better resource utilization. Our results demonstrate that virtualization technology coupled with lightweight online scheduling strategies can afford dramatic improvements in performance for executing HPC workloads.

Index Terms—cluster, scheduler, virtual machine, vector bin packing, high performance computing, batch scheduling

1 Introduction

The standard method for sharing a cluster among High Performance Computing (HPC) users is batch scheduling. With batch scheduling, users submit *requests* to run applications, or *jobs*. Each request is placed in a queue and waits to be granted an *allocation*, that is, a subset of the cluster's compute nodes, or *nodes* for short. The job has exclusive access to these nodes for a bounded duration.

One problem with batch scheduling is that it inherently limits overall resource utilization. If a job uses only a fraction of a node's resource (e.g., half of the processor cores, a third of the memory), then the remainder of it is wasted. It turns out that this is the case for many jobs in HPC workloads [1], [2]. Additionally, since batch schedulers use integral resource allocations with no time-sharing of nodes, incoming jobs can be postponed even while some nodes are sitting idle.

A second problem is the known disconnect with user concerns (response time, fairness) [3]. While batch schedulers provide myriad configuration parameters, these parameters are not directly related to relevant user-centric metrics.

In this work we seek to remedy both of the above problems. We address the first by allowing fractional resource allocations (e.g., allocating 70% of a resource to a job task) that can be modified on the fly (e.g., by changing allocated fractions, by migrating job tasks to different nodes). We address the second by defining an objective performance metric and developing algorithms that attempt to optimize it.

Existing optimization approaches generally assume that job processing times are known [4] or that reliable estimates are available [5]. Unfortunately, job processing time estimates are often wildly inaccurate [6]. We take a drastic departure from the literature and assume no knowledge of job processing times.

Our approach, which we term *dynamic fractional resource scheduling* (DFRS), amounts to a carefully controlled time-sharing scheme enabled by virtual machine (VM) technology. Other VM-based scheduling approaches have been proposed, but the research in this area has focused primarily on technical issues [7] or extensions to existing scheduling schemes, such as combining best-effort and reservation based jobs [8]. In this paper we:

- Define the offline and online DFRS problems and establish their theoretical difficulty;
- Derive absolute performance bounds for any given problem instance;
- Propose algorithms for solving the online non-clairvoyant DFRS problem;
- Evaluate our algorithms in simulation using synthetic and real-world HPC workloads;
- Identify algorithms that outperform batch scheduling by orders of magnitude;
- Define a new metric to capture the notion of efficient resource utilization;
- Demonstrate that our algorithms can be easily tuned so that they are as, or more, efficient than batch scheduling while still outperforming it by orders of magnitude.

We formalize the DFRS problem in Section 2, study its theoretical difficulty in Section 3, and propose DFRS algorithms in Section 4. We describe our experimental methodology and present results in Section 5. We conclude with a summary of results and a highlights of future directions in Section 6.

- *M. Stillwell was formerly with the Department of Information and Computer Sciences, University of Hawai'i at Mānoa, Honolulu, U.S.A. and is currently with INRIA in Lyon, France.*
- *H. Casanova is with the Department of Information and Computer Sciences, University of Hawai'i at Mānoa, Honolulu, U.S.A*
- *F. Vivien is with INRIA, the Université de Lyon, and the LIP laboratory, UMR 5668 ENS-CNRS-INRIA-UCBL, in Lyon, France.*

This work was partially supported by the U.S. National Science Foundation under award #0546688, and by the European Commission's Marie Curie Fellowship IOF #221002.

2 THE DFRS APPROACH

DFRS requires fractional allocation of resources, such as CPU cycles, and thus uses time-sharing. The classical time-sharing solution for parallel jobs is gang scheduling [9], which, because of its drawbacks, is used far less often than batch scheduling for HPC clusters (see Section 6 of the web supplement). In this work we opt for time-sharing in an uncoordinated and low-overhead manner, as enabled by virtual machine (VM) technology (see Section 1 of the web supplement). Our goal is to circumvent the aforementioned problems of batch scheduling without suffering from the drawbacks of gang scheduling.

We consider a homogeneous cluster based on a switched interconnect with network-attached storage. Users submit requests to run jobs that consist of one or more tasks to be executed in parallel. Each task runs within a VM instance. Our goal is to make sound resource allocation decisions. These decisions include selecting initial nodes for VM instances, setting allocated resource fractions for each instance, migrating instances between nodes, preempting and pausing instances (by saving them to local or network-attached storage), and postponing incoming job requests.

Each task has a *memory requirement*, expressed as a fraction of node memory, and a *CPU need*, which is the fraction of node CPU cycles that the task needs to run at maximum speed. Memory capacities of nodes should not be exceeded. This is to avoid the use of process swapping, which can have a hard to predict but almost always dramatic impact on task execution times. We do, however, allow for overloading of CPU resources. Further, the CPU fraction actually allocated to the task can change over time, e.g., it may need to be decreased due to the system becoming more heavily loaded. When a task is given less than its CPU need we assume that its execution is slowed proportionally. The task then completes once the cumulative CPU resource assigned to it up to the current time is equal to the product of its need and execution time on a dedicated system. Note that a task can never be allocated more CPU than its need. We target HPC workloads and assume that all tasks in a job have the same memory requirements and CPU needs, and that they must progress at the same rate.

One metric commonly used to evaluate batch schedules is the *stretch* (or slowdown) [10]. The stretch of a job is defined as its actual turn-around time divided by its turn-around time had it been alone on the cluster. For instance, a job that could have run in 2 hours on the dedicated cluster but instead runs in 4 hours due to competition with other jobs experiences a stretch of 2. In the literature a proposed way to optimize both for aggregate performance and for fairness is to minimize the maximum stretch [10], as opposed to simply minimizing average stretch, the latter being prone to starvation [11]. Maximum stretch minimization is known to be theoretically difficult, and even in clairvoyant settings there does not exist any constant-ratio competitive algorithm [11]. Nevertheless, heuristics can lead to good results in practice [11].

Stretch minimization, and especially maximum stretch minimization, tends to favor short jobs, but on real clusters the jobs with the shortest running times are often those that fail at launch time. To prevent our evaluation of schedule quality from being dominated by these faulty jobs, we adopt a variant of the stretch called the *bounded stretch*, or "bounded slowdown" [9]. In this variant, the turn-around time of a job is replaced by a threshold value if this turn-around time is smaller than that threshold. We set the threshold to 10 seconds, and hereafter we use the term stretch to mean bounded stretch.

In this work we do not assume *any* knowledge of job processing times as these estimates are typically (wildly) inaccurate [6]. Relying on them is thus a losing proposition. Instead, we define a new metric, the *yield*, that does not use job processing time estimates. The *yield* of a task is the instantaneous fraction of the CPU resource of the node allocated to the task divided by the task's CPU need (to a maximum of 1). Since we assume that all tasks within a job have identical CPU needs and are allocated identical CPU fractions, they all have the same yield which is then the yield of the job.

Our goal is to develop algorithms that explicitly seek to maximize the minimum yield, which is closely related to the maximum stretch. The key questions are whether this strategy will lead to good stretch values in practice, and whether DFRS will be able to outperform standard batch scheduling algorithms.

3 THEORETICAL ANALYSIS

In this section we study the offline scenario so that we can derive a lower bound on the optimal maximum stretch of any instance assuming a clairvoyant scenario. We then quantify the difficulty of the online, non-clairvoyant case. Even in an offline scenario and even when ignoring CPU needs, memory constraints make the problem of maximum stretch minimization NP-hard in the strong sense since it becomes a bin-packing problem. Consequently, in this section we assume that all jobs have null memory requirements, or, conversely, that memory resources are infinite.

Formally, an instance of the offline problem is defined by a set of jobs, $\mathcal{J}$, and a set of nodes, $\mathcal{P}$. Each job j has a set, $\mathcal{T}_j$, of tasks and a CPU need, c_j, between 0 and 1. It is submitted at its *release date* r_j and has *processing time* p_j, representing its execution time on an equivalent dedicated system. A target value $\mathcal{S}$ for the maximum stretch defines a deadline $d_j = r_j + \mathcal{S} \times p_j$ for the execution of each job j. The set of job release dates and deadlines, $\mathbb{D} = \bigcup_{j \in \mathcal{J}} \{r_j, d_j\}$, gives rise naturally to a set $\mathcal{I}$ of consecutive, non-overlapping, left-closed intervals that cover the time span $[\min_{j \in \mathcal{J}} r_j, \max_{j \in \mathcal{J}} d_j)$, where the upper and lower bounds of each interval in $\mathcal{I}$ are members of $\mathbb{D}$ and each member of $\mathbb{D}$ is either the upper or lower bound of at least one member of $\mathcal{I}$. For any interval t we define $\ell(t) = \sup t - \inf t$ (i.e., $\ell(t)$ is the *length* of t).

A *schedule* is an allocation of processor resources to job tasks over time. For a schedule to be valid it must satisfy the following conditions: 1) every task of every job j must receive $c_j \times p_j$ units of work over the course of the schedule, 2) no task can begin before the release date of its job, 3) at any given moment in time, no more than 100% of any CPU can be allocated to running tasks, 4) the schedule can be broken down

into a sequence of consecutive, non-overlapping time spans no larger than some time quantum $\mathcal{Q}$, such that over each time span every task of a job j receives the same amount of work and each of these tasks receives no more than c_j times the length of the time span units of work. Within these small time spans any task can fully utilize a CPU resource, regardless of the CPU need of its job, and the tasks of a job can proceed independently. The exact size of $\mathcal{Q}$ depends upon the system, but as the timescale of parallel job scheduling is orders of magnitude larger than that of local process scheduling, we make the reasonable assumption that for every $t \in \mathcal{I}$, $\mathcal{Q} << \ell(t)$.

Theorem 1. *Let us consider a system with infinite memory and assume that any task can be moved instantaneously and without penalty from one node to another. Then there exists a valid schedule whose maximum stretch is no greater than $\mathcal{S}$ if and only if the following linear system has a solution, where each variable α_j^t represents the portion of job j completed in time interval t:*

$$\begin{cases} \text{(1a)} \quad \forall j \in \mathcal{J} & \sum_{t \in \mathcal{I}} \alpha_j^t = 1; \\ \text{(1b)} \quad \forall j \in \mathcal{J}, \forall t \in \mathcal{I} & r_j \geq \sup t \Rightarrow \alpha_j^t = 0; \\ \text{(1c)} \quad \forall j \in \mathcal{J}, \forall t \in \mathcal{I} & d_j \leq \inf t \Rightarrow \alpha_j^t = 0; \\ \text{(1d)} \quad \forall j \in \mathcal{J}, \forall t \in \mathcal{I} & \alpha_j^t p_j \leq \ell(t); \\ \text{(1e)} \quad \forall t \in \mathcal{I} & \sum_{j \in \mathcal{J}} \alpha_j^t p_j c_j |\mathcal{T}_j| \leq |\mathcal{P}| \ell(t). \end{cases} \tag{1}$$

Since all variables of Linear System (1) are rational, one can check in polynomial time whether this system has a solution. Using a binary search, one can then use the above theorem to find a lower bound on the optimal maximum stretch in polynomial time (see Section 2.1 of the web supplement for proof and details).

Online maximum stretch minimization is known to be theoretically difficult. Recall that the competitive ratio of an online algorithm is the minimum worst-case ratio of the performance of that algorithm with the optimal offline algorithm. Even in a clairvoyant scenario there is no constant-ratio competitive online algorithm [11]. Further, we have established that the number of jobs under consideration is a lower bound on the competitive ratio. A second independent lower bound can be expressed as $\frac{1}{2}\Delta^{\sqrt{2}-1}$, where Δ is the ratio of the processing time of the longest job to that of the shortest. Details and proofs are provided in Section 2.2 of the web supplement.

4 DFRS Algorithms

Our theoretical results demonstrate that non-clairvoyant maximum stretch optimization is "hopeless": no algorithm can be designed with a low worst-case competitive ratio because of the large number of jobs and/or the large ratio between the largest and smallest jobs found in HPC workloads. Instead, we focus on developing non-guaranteed algorithms (i.e., heuristics) that perform well in practice, hopefully close to the offline bound.

We propose to adapt the algorithms we designed in our study of the offline resource allocation problem for static workloads [12], [13]. Due to memory constraints, it may not always be possible to schedule all currently pending jobs simultaneously; It is therefore necessary to establish a measure of priority among jobs. In this section we define and justify our job priority function (Section 4.1), describe the greedy (Section 4.2) and vector packing based (Section 4.3) task placement heuristics, explain how these heuristics can be combined to create heuristics for the online problem (Sections 4.4 and 4.5), give our basic strategies for resource allocation once tasks are mapped to nodes (Section 4.6), and finally provide an alternative algorithm that attempts to optimize stretch directly instead of relying on the yield (Section 4.7).

4.1 Prioritizing Jobs

We define a priority based on the *virtual time* of a job, that is, the total subjective execution time experienced by a job. Formally, this is the integral of the job's yield since its release date. For example, a job that starts and runs for 10 seconds with a yield of 1.0, that is then paused for 2 minutes, and then restarts and runs for 30 seconds with a yield 0.5 has experienced 25 total seconds of virtual time ($10 \times 1.0 + 120 \times 0.0 + 30 \times 0.5$). An intuitive choice for the priority function is the inverse of the virtual time: the shorter the virtual time, the higher the priority. A job that has not yet been allocated any CPU time has a zero virtual time, i.e., an infinite priority.

This approach, however, has a prohibitive drawback: The priority of paused jobs remains constant, which can induce starvation. Thus, the priority function should also consider the *flow time* of a job, i.e., the time elapsed since its submission. This would prevent starvation by ensuring that the priority of any paused job increases with time and tends to infinity.

Preliminary experimental results showed that using the inverse of the virtual time as a priority function leads to good performance, while using the ratio of flow time to virtual time leads to poor performance. We believe that this poor performance is due to the priorities of jobs all converging to some constant value related to the average load on the system. As a result, short-running jobs, which suffer a greater penalty to their stretch when paused, are not sufficiently prioritized over longer-running jobs. Consequently, we define the priority function as: priority $= \frac{\text{flow time}}{(\text{virtual time})^2}$. The power of two is used as a heuristic to increase the importance of the virtual time with respect to the flow time, thereby giving an advantage to short-running jobs. We break ties between equal-priority jobs by considering their order of submission.

4.2 Greedy Task Mapping

A basic greedy algorithm, which we simply call Greedy, allocates nodes to an incoming job j without changing the mapping of tasks that may currently be running. It first identifies the nodes that have sufficient available memory to run at least one task of job j. For each of these nodes it computes its *CPU load* as the sum of the CPU needs of all the tasks currently allocated to it. It then assigns one task of job j to the node with the lowest CPU load, thereby picking

the node that can lead to the largest yield for the task. This process is repeated until all tasks of job j have been allocated to nodes, if possible. A weakness of this algorithm is that it can allow a newly submitted short-running job to be stalled indefinitely, leading to unbounded values for maximum stretch.

In order to address this problem we define two variants of Greedy that make use of the priority function as defined previously. GreedyP is like Greedy except that it can pause some running jobs in favor of newly submitted jobs. Jobs are selected for pausing based on their priority and whether or not doing so would allow the newly submitted job to start. GreedyPM extends GreedyP with the capability of moving rather than pausing running jobs. This is done by trying to reschedule jobs selected for pausing in order of their priority using Greedy.

4.3 Task Mapping as Vector Packing

The Greedy algorithm builds a solution through a succession of locally optimal decisions, but the final solution may be far from globally optimal. An alternative approach is to compute a global solution from scratch and then preempt and/or migrate tasks if necessary. As we have two resource dimensions (CPU and memory), our resource allocation problem is related to a version of bin packing, known as two-dimensional *vector packing*. One important difference between our problem and vector packing is that our jobs have fluid CPU needs. This difference can be addressed as follows: Consider a fixed value of the yield, Y, that must be achieved for all jobs. Fixing Y amounts to transforming all CPU needs into *CPU requirements*: simply multiply each CPU need by Y. The problem then becomes exactly vector packing and we can apply a preexisting heuristic to solve it. We use a binary search on Y to find the highest yield for which the vector packing problem can be solved (our binary search has an accuracy threshold of 0.01).

In previous work [13] we developed an algorithm based on this principle called MCB8. It makes use of a multi-dimensional vector packing heuristic based on that described by Leinberger et al. in [14]. A detailed description of how the algorithm functions when applied to a two-dimensional vector packing problem is given in Section 3.1 of the web supplement. In the event that MCB8 cannot find a valid allocation for all of the jobs currently in the system at any yield value, it removes the lowest priority job from consideration and tries again.

Limiting Migration

The preemption or migration of newly-submitted short-running jobs can lead to poor performance. To mitigate this behavior, we introduce two parameters. If set, the MINFT parameter (respectively, the MINVT parameter), stipulates that jobs whose flow-times (resp., virtual times) are smaller than a given bound may be paused in order to run higher priority jobs, but, if they continue running, their current node mapping must be maintained. Jobs whose flow-times (resp., virtual times) are greater than the specified bound may be moved as previously. Migrations initiated by GreedyPM are not affected by these parameters.

4.4 When to Compute New Task Mappings

The most obvious times to apply our task mapping algorithms are when a new job is submitted to the system and when a running job completes and exits. The MCB8 algorithm attempts a global optimization and, thus, can (theoretically) "reshuffle" the whole mapping each time it is invoked[1]. One may thus fear that applying MCB8 on each submission could lead to a prohibitive number of preemptions and migrations. On the contrary, Greedy has low overhead and the addition of a new job should not be overly disruptive to currently running jobs. The counterpart is that Greedy may generate allocations that use cluster resources inefficiently. For both of these reasons we experiment and consider algorithms that: upon job submission, either do nothing or apply Greedy, GreedyP, GreedyPM, or MCB8; upon job completion, either do nothing or apply Greedy or MCB8; and either apply or do not apply MCB8 periodically.

4.5 Algorithm Naming Scheme

We use a multi-part scheme for naming our algorithms, using '/' to separate the parts. The first part corresponds to the policy used for scheduling jobs upon submission, followed by a '*' if jobs are also scheduled opportunistically upon job completion (using MCB8 if MCB8 was used on submission, and Greedy if Greedy, GreedyP, or GreedyPM was used). If the algorithm applies MCB8 periodically, the second part contains "per". For example, the GreedyP*/per algorithm performs a Greedy allocation with preemption upon job submission, opportunistically tries to start currently paused jobs using Greedy whenever a job completes, and periodically applies MCB8. The use of our four different scheduling strategies upon job submission, combined with the use of scheduling paused jobs on job completion, periodically rescheduling jobs using MCB8, or doing both, creates 12 combinations of algorithms. The periodic use of MCB8 without scheduling jobs on submission or completion adds another possibility, for a total of 13 basic algorithms. Parameters, such as MINVT or MINFT, are appended to the name of the algorithm if set.

4.6 Resource Allocation

Once tasks have been mapped to nodes one has to decide on a CPU allocation for each job (all tasks in a job are given identical CPU allocations). All previously described algorithms use the following procedure: First all jobs are assigned yield values of $1/\max(1, \Lambda)$, where Λ is the maximum CPU load over all nodes. This maximizes the minimum yield given the mapping of tasks to nodes. We use two different approaches to exploit any remaining resource fractions.

Average Yield Optimization

We can use a rational linear program to find a resource allocation that maximizes the average yield under the constraint that no job is given a yield lower than the maximized minimum. Algorithms that use this optimization have "OPT=AVG" as an additional part of their names.

1. In practice this does not happen because the algorithm is deterministic and always considers the tasks and the nodes in the same order.

Max-min Yield Optimization

As an alternative to maximizing the average yield, we consider the iterative maximization of the minimum yield. At each step the minimum yield is maximized using the procedure described at the beginning of Section 4.6. Those jobs whose yield cannot be further improved are removed from consideration, and the minimum is further improved for the remaining jobs. This process continues until no more jobs can be improved. This type of max-min optimization is commonly used to allocate bandwidth to competing network flows [15, Chapter 6]. Algorithms that use this optimization have "OPT=MIN" as an additional part of their names.

4.7 Optimizing the Stretch Directly

All algorithms described thus far attempt to optimize the minimum yield in the hope that this strategy will result in low values for the maximum stretch. We also consider a variant of /per, called /stretch-per, that tries to minimize the maximum stretch directly, still assuming no knowledge of job processing times. The algorithm it uses, called MCB8-stretch, can only be applied periodically as it is based on knowledge of the time between scheduling events. It follows the same general procedure as MCB8 but focuses on minimizing the maximum stretch, assuming that no jobs terminate before the next scheduling event. For the resource allocation improvement phase we use algorithms similar to those described in Section 4.6, except that the first (OPT=AVG) seeks to minimize the average expected stretch and the second (OPT=MAX) iteratively minimizes the maximum expected stretch.

5 Experimental Results

We use a discrete event simulator to compare the performance of our algorithms with conventional batch scheduling approaches on three sets of traces. The batch scheduling algorithms under consideration are first-come-first-served (FCFS) and the currently standard EASY [9]. The first trace set is composed of 100 synthetic traces generated using a well established model [16]; each contains 1,000 jobs submitted over a period of 4-6 days to a cluster with 128 quad-core nodes. To make the second set of traces, those in the first set were scaled to produce 9 distinct offered load levels of 0.1 to 0.9. These two synthetic trace sets are thus referred to as the "unscaled synthetic traces" and "scaled synthetic traces". The third trace set is made up of the weekly submission activity to the HPC2N cluster, which is composed of 120 dual-core nodes, over the course of 182 weeks [1]. The individual traces in all three sets were annotated with CPU needs and memory requirements as described in Section 4 of the web supplement. In order to demonstrate that our algorithms will prove beneficial even in non-ideal conditions, we conservatively assume a five-minute penalty for each time a job is paused or one of its tasks is migrated. Considering the capabilities of modern high-end systems we believe that this represents a conservative overestimation of the real-world performance penalty.

5.1 Stretch Results

For a given problem instance, and for each algorithm, we define the *degradation from bound* as the ratio between the maximum stretch achieved by the algorithm on the instance and the theoretical lower bound on the optimal maximum stretch obtained using the offline algorithm described in Section 3. A lower value of this ratio thus denotes better performance. We consider average and standard deviation values computed over sets of problem instances, i.e., for each of our set of traces. We also determine maximum values, i.e., the result for the "worst trace" for each algorithm.

Fourteen combinations of mechanisms for mapping tasks to processors were described in Section 4 (Table 1 of the web supplement gives the complete list). Each combination can use either OPT=AVG or OPT=MIN to compute resource allocations once a mapping has been determined. Furthermore, eleven of these combinations use the MCB8 algorithm and thus can optionally use either the MINFT or MINVT parameter. Even when limiting the discussion to MINFT=300, MINFT=600, MINVT=300 and MINVT=600, the total number of potential algorithms is $3 \times 2 + 11 \times 2 \times 5 = 116$. However, the full results (available in Appendix A of the web supplement) show that on average OPT=MIN is never worse and often slightly better than OPT=AVG. Consequently, we consider results only for algorithms that use OPT=MIN. Furthermore, we find that among the mechanisms for limiting task remapping, MINVT is always better than MINFT, and slightly better with the larger 600s bound.

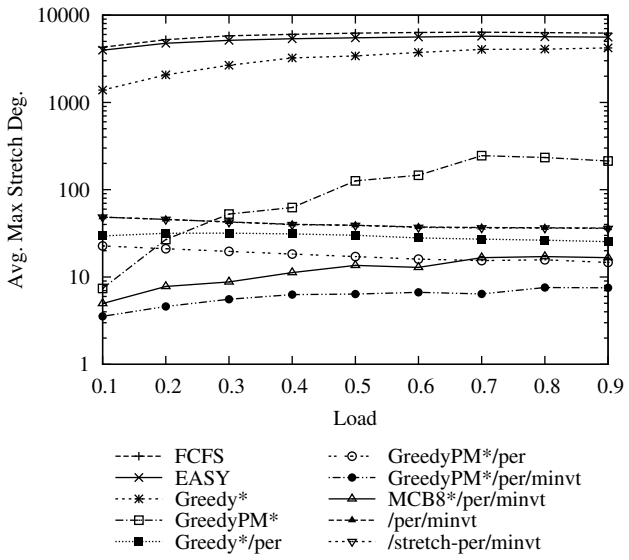

Figure 1. Average degradation from bound vs. load for selected algorithms on the scaled synthetic dataset. All algorithms use OPT=MIN (except /stretch-per, which uses OPT=MAX) and use MINVT=600 if MINVT is specified.

Figure 1 plots average degradation factors vs. load for selected algorithms when applied to scaled synthetic workloads using a logarithmic scale on the vertical axis. The key observation from the figure is that EASY and FCFS are almost always outperformed by our proposed algorithms by

several orders of magnitude, thereby showing that DFRS is an attractive alternative to batch scheduling. The only DFRS algorithm that shows poor performance is Greedy*/OPT=MIN, which, as explained in Section 4.2, can lead to high stretch for short-running jobs. Another notable feature is that the GreedyPM*/OPT=MIN algorithm performs well under low load conditions, but quickly leads to poorer performance than most other algorithms as the load becomes larger than 0.3. Under low-load conditions, the periodic use of MCB8 to remap tasks is not as critical to ensure good performance. Overall, the best DFRS algorithm regardless of the load level is GreedyPM*/per/OPT=MIN/MINVT=600. The relative performance of all algorithms remains consistent across all three sets of traces, with a few exceptions. See Section 5.1 of the web supplement for a more complete discussion.

Our overall conclusion is that, to achieve the best performance, all our techniques should be combined: an aggressive greedy job admission policy, a periodic use of the MCB8 vector-packing algorithm, an opportunistic use of resources freed upon job completion, and a grace period that prevents remapping of tasks that have just begun executing. Note that while the algorithms are executed in an online, non-clairvoyant context, the computation of the bound on the optimal performance relies on knowledge of both the release dates and processing times of all jobs. Furthermore, the bound ignores memory constraints. Nevertheless, in our experiments, our best algorithms are on average at most a factor 7 away from this loose bound. We conclude that our algorithms lead to good performance in an absolute sense.

5.2 Impact of Preemptions and Migrations

The main observation from our results is that the bandwidth required for DFRS is reasonable. The GreedyPM*/per/OPT=MIN/MINVT=600 algorithm uses under 0.80 GB/sec on average for the set of "heavy workloads" (scaled synthetic traces with load values ≥ 0.7). Even accounting for maximum bandwidth consumption, i.e., for the trace that causes the most traffic due to preemptions and migrations, bandwidth consumption is under 2.0 GB/sec. Such numbers represent only a small fraction of the bandwidth capacity of current cluster platforms. This algorithm initiates fewer than 38 preemptions and 53 migrations each hour, with each job being preempted under 6 times and migrating under 7 times during its lifetime, on average. A more complete discussion of the migration costs can be found in Section 5.3 of the web supplement.

5.3 Platform Utilization

In this section we investigate how our best algorithms, in terms of maximum stretch, compare to EASY in terms of platform utilization. We introduce a new metric, called *underutilization*. We contend that this metric helps quantify schedule quality in terms of utilization, while remaining agnostic to conditions that can confound traditional metrics, as explained hereafter.

5.3.1 Measuring System Underutilization

A common concern of cluster administrators is to maximize platform utilization so as to both improve throughput and justify the costs associated with maintaining a cluster. Metrics used to evaluate machine utilization include throughput, daily peak instantaneous utilization, and average instantaneous utilization over time. However, these metrics are not appropriate for open, online systems as they are highly dependent on the arrival process and the requirements of jobs in the workload [17]. Another potential candidate is the makespan, i.e., the amount of time elapsed between the submission of the first job in the workload and the completion of last job. This metric suffers from the problem that a very short job may be submitted at the last possible instant, resulting in all scheduling algorithms leading to (nearly) the same makespan [17].

Instead, we argue that the quality of a scheduling algorithm (not considering fairness to jobs) should be judged based on how well it meets demand over the course of time, bounded by the resource constraints of the system. We call our measure of this quantity the *underutilization* and define it as follows. For a fixed set of nodes $\mathcal{P}$, let $D^{\sigma}_{\mathcal{J}}(t)$ be the total CPU demand (i.e., sum of CPU needs) on resources by jobs from $\mathcal{J}$ that have been submitted but have not yet completed at time t when scheduled using algorithm σ, and let $u^{\sigma}_{\mathcal{J}}(t)$ represent the total system utilization at time t (i.e., sum of allocated CPU fractions) under the same conditions. The underutilization of a system (assuming that all release dates are ≥ 0) is given by $\int_0^{\infty} \min\{|\mathcal{P}|, D^{\sigma}_{\mathcal{J}}(t)\} - u^{\sigma}_{\mathcal{J}}(t)dt$. Note that $u^{\sigma}_{\mathcal{J}}(t)$ is constrained to always be less than both $|\mathcal{P}|$ and $D^{\sigma}_{\mathcal{J}}(t)$, and that $D^{\sigma}_{\mathcal{J}}(t) = 0$ outside of the time span between when the first job is submitted and the last one completes.

Essentially, underutilization represents a cumulative measure over time of computational power that could, at least theoretically, be exploited, but that is instead sitting idle. Thus, lower values for underutilization are preferable to higher values. As this quantity depends on total workload demand, which can vary considerably, when combining results over a number of workloads we consider the normalized underutilization, or the underutilization as a fraction of the total resources required to execute the workload. We contend that, for a given platform, algorithms that do a better job of allocating resources will tend to have smaller values for normalized underutilization in the average case on a given set of workloads. It is important to note that normalized underutilization will vary with platform and workload characteristics, and thus it should not be taken as an absolute measure of algorithm efficiency.

5.3.2 Experimental Results

In this section we consider only the EASY batch scheduling algorithm and the GreedyP*/per/OPT=MIN/MINVT=600 and GreedyPM*/per/OPT=MIN/MINVT=600 algorithms, which are the best algorithms identified in Section 5.1. We use the same basic experimental setup as described in Section 5, including the 5-minute penalty for preempting or migrating jobs.

On the real-world trace EASY leads to an average normalized underutilization value of 6.4%, which is quite low, while our algorithms both score 34.4%. Our algorithms do somewhat better on the synthetic traces, scoring 49.7% each to EASY's 34.9% on the unscaled set, and each scoring just under 61.0% on the scaled set of traces to EASY's 38.4%.

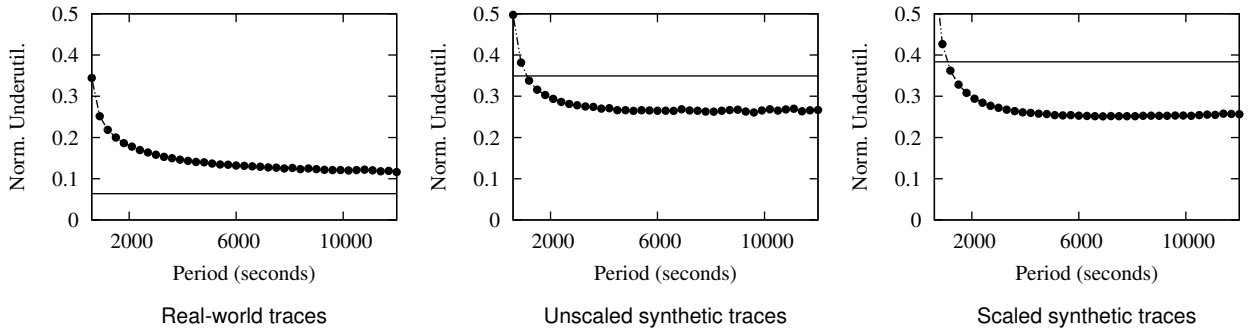

Figure 2. Avg. normalized underutilization vs. period for EASY (solid) and GreedyPM*/per/OPT=MIN/MINVT=600 (dots)

While EASY scores better on this initial comparison, there are several factors to consider. The first is that the very low underutilization shown by EASY against the real-world trace may be partially due to an artifact of either user or system behavior (e.g., the system from which the log was harvested also made use of the EASY scheduler). Also for the real trace, the fact that we arbitrarily split the entire HPC2N trace into 182 week-long periods may be a factor. A simulation for the full trace yields a normalized underutilization of 8% for EASY and of 10% for our algorithms.

Still, our proposed algorithms also perform worse in terms of underutilization on the synthetic traces, which should be free from both of the above problems. We hypothesize that this lower efficiency is caused by time spent doing preemption and migration. Recall that all our algorithms use a 600 second period, equal to the rescheduling penalty, by default. By increasing the period, within reasonable bounds, one can then hope to decrease underutilization. The trade-off, however, is that the maximum stretch may be increased.

Figure 2 shows underutilization results for our three sets of traces for the GreedyPM*/per/OPT=MIN/MINVT=600 algorithm (results for GreedyP*/per/OPT=MIN/MINVT=600 are similar). For each set, we plot the average normalized underutilization versus the period. In all graphs the period varies from 600 to 12,000 seconds, i.e., from 2x to 20x the rescheduling penalty. We do not use a period equal to the penalty as for some traces it can lead to job thrashing. In all graphs normalized underutilization is shown to decrease steadily. Results available in Appendix B of the web supplement show that for extremely large periods (over 15,000 seconds for the synthetic traces) underutilization expectedly begins to increase. For the two sets of synthetic traces, as the period becomes larger than 900s (i.e., 1.5x the rescheduling penalty), our algorithms achieve better average normalized underutilization than EASY. For the real-world trace, our algorithms achieve higher values than EASY regardless of the period: EASY achieves low values at around 6.4%, while our algorithm plateaus at around 11.8%, before slowly starting to increase again for the largest periods that we studied.

Figure 3 shows the average maximum stretch for our algorithm as the period increases, for each set of traces. The main observation is that the average maximum stretch degradation increases slowly as the period increases. The largest increase is seen for the scaled synthetic traces, for which a 20-fold

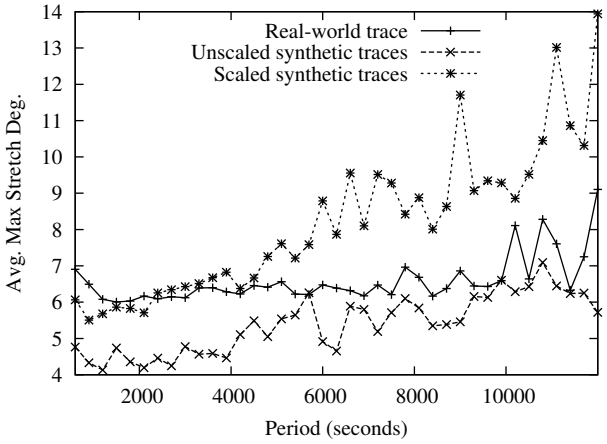

Figure 3. Max stretch deg. from bound vs. period for GreedyPM*/per/OPT=MIN/MINVT=600 for all trace sets

increase of the period leads to an increase of the maximum stretch degradation by less than a factor of 3. Results for the unscaled synthetic traces and the real-world trace show much smaller increases (by less than a factor 1.5 and a factor 2, respectively). Consequently, increasing the period significantly, i.e., up to 20x the rescheduling penalty, still allows our algorithms to outperform EASY by at least two orders of magnitude on average.

We conclude that our algorithms can outperform EASY by orders of magnitude in terms of maximum stretch, and lead to only slightly higher or significantly lower underutilization provided the period is set appropriately. Our results indicate that picking a period roughly between 5x and 20x the rescheduling penalty leads to a good compromise between performance and efficiency. Our approach is thus robust to the choice of the period, and in practice a period of, say, 1 hour, is appropriate. Results not included here show that with a 1-hour period the bandwidth consumption due to preemptions and migrations is roughly 4 times lower than that reported in Section 5.2, e.g., under 0.2GB/sec on the average when considering the scaled synthetic traces with load values ≥ 0.7 (see Figure 5 in the web supplement). In conclusion, we recommend using the GreedyPM*/per/OPT=MIN/MINVT=600 algorithm with a period equal to 10 times the rescheduling penalty.

6 Conclusion

In this paper we have proposed DFRS, a novel approach for job scheduling on a homogeneous cluster. We have focused on an online, non-clairvoyant scenario in which job processing times are unknown ahead of time. We have proposed several scheduling algorithms and have compared them to standard batch scheduling approaches using both real-world and synthetic workloads. We have found that several DFRS algorithms lead to dramatic improvement over batch scheduling in terms of maximum (bounded) stretch. In particular, algorithms that periodically apply the MCB8 vector packing based algorithm lead to the best results. Our results also show that the network bandwidth consumption of these algorithms for job preemptions and migrations is only a small fraction of that available in current clusters. Finally, we have shown that these algorithms can lead to good platform utilization as long as the period at which MCB8 is applied is chosen within a broad range. The improvements shown in our results are likely to be larger in practice due the many conservative assumptions in our evaluation methodology.

This work opens a number of promising directions for future research. Our scheduling algorithms could be improved with a strategy for reducing the yield of long running jobs. This strategy, inspired by thread scheduling in operating systems kernels, would be useful for mitigating the negative impact of long running jobs on shorter ones, thereby improving fairness. While we considered the case for HPC jobs composed of tasks with homogeneous resource requirements and needs, the techniques that we developed could easily be modified to allow for heterogeneous tasks as well (see our paper on the offline problem [12] for an expanded discussion of this issue). Also, mechanisms for implementing user priorities, such as those supported in batch scheduling systems, are needed. More broadly, a logical next step is to implement our algorithms as part of a prototype virtual cluster management system.

Acknowledgment

Simulations were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] D. G. Feitelson, "Parallel workloads archive." [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/

[2] S. K. Setia, M. S. Squillante, and V. K. Naik, "The impact of job memory requirements on gang-scheduling performance," *ACM SIGMETRICS Perf. Eval. Rev.*, vol. 26, no. 4, pp. 30–39, 1999.

[3] C. B. Lee and A. E. Snavely, "Precise and realistic utility functions for user-centric performance analysis of schedulers," in *HPDC*, 2007, pp. 107–116.

[4] M. A. Bender, S. Muthukrishnan, and R. Rajaraman, "Approximation algorithms for average stretch scheduling," *J. Scheduling*, vol. 7, no. 3, pp. 195–222, 2004.

[5] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," in *ICPP Wksp.*, 2002, pp. 514–522.

[6] C. B. Lee and A. E. Snavely, "On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions," *Intl. J. HPC Apps.*, vol. 20, no. 4, pp. 495–506, 2006.

[7] N. Bhatia and J. S. Vetter, "Virtual cluster management with Xen," in *Euro-Par Wksp.*, 2007, pp. 185–194.

[8] B. Sotomayor, K. Keahey, and I. Foster, "Combining batch execution and leasing using virtual machines," in *HPDC*, 2008, pp. 87–96.

[9] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *JSSPP*, 1997, pp. 1–34.

[10] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and stretch metrics for scheduling continuous job streams," in *ACM-SIAM SODA*, 1998, pp. 270–279.

[11] A. Legrand, A. Su, and F. Vivien, "Minimizing the stretch when scheduling flows of divisible requests," *J. Scheduling*, vol. 11, no. 5, pp. 381–404, 2008.

[12] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *JPDC*, vol. 70, no. 9, pp. 962–974, 2010.

[13] ——, "Resource allocation using virtual clusters," in *CCGrid*, 2009, pp. 260–267.

[14] W. J. Leinberger, G. Karypis, and V. Kumar, "Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints," in *ICPP*, 1999, pp. 404–412.

[15] D. P. Bertsekas and R. Gallager, *Data Networks*, 2nd ed. Prentice Hall, 1992.

[16] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs," *JPDC*, vol. 63, no. 11, 2003.

[17] E. Frachtenberg and D. G. Feitelson, "Pitfalls in parallel job scheduling evaluation," in *JSSPP*, 2005, pp. 257–282.

Author Information

Mark Stillwell is an INRIA postdoctoral researcher at the École Normale Supérieure de Lyon. He obtained his Ph.D. from the University of Hawai'i at Mānoa, and previously received Masters degrees in mathematics and computer science from the University of Florida. He has had one other paper published in an international journal and 3 papers published in international conferences. His research interests include job scheduling, parallel algorithms, and computational complexity.

Frédéric Vivien received a Ph.D. degree from École Normale Supérieure de Lyon in 1997. From 1998 to 2002, he was an associate professor at Louis Pasteur University, in Strasbourg, France. He spent the year 2000 working in the Computer Architecture Group of the MIT Laboratory for Computer Science. He is currently a full researcher at INRIA, working at the ENS Lyon, France. He visited for one year the CoRG group of the University of Hawai'i at Mānoa. He is the author of one book, 25+ papers published in international journals, and 40+ papers published in international conferences. He is also the editor of one textbook. His main research interests are scheduling techniques and parallel algorithms for distributed and/or heterogeneous systems.

Henri Casanova is an Associate Professor in the Information and Computer Science Dept. at the University of Hawai'i at Mānoa. He obtained his M.S. from the Université Paul Sabatier, Toulouse, France in 1994, and his Ph.D. from the University of Tennessee, Knoxville in 1998. Prior to joining the University of Hawai'i, he was a Research Scientist at the San Diego Supercomputer Center and an Adjunct Professor in the Dept. of Computer Science and Engineering at the University of California, San Diego. He has co-authored one graduate-level textbook on the topic of parallel algorithm, 35+ articles in archival journals, and 65+ articles in international conferences. His research interests are in the area of parallel and distributed computing.