International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume X, Number Y, pages 1–25, January 2020

Resilient Scheduling Heuristics for Rigid Parallel Jobs

Anne Benoit
ENS Lyon, CNRS & Inria, France

Valentin Le Fèvre
ENS Lyon, CNRS & Inria, France

Padma Raghavan
Vanderbilt University, Nashville, TN, USA

Yves Robert
ENS Lyon, CNRS & Inria, France
University of Tennessee Knoxville, TN, USA

Hongyang Sun
Vanderbilt University, Nashville, TN, USA

Received: (received date)
Revised: (revised date)
Accepted: (accepted date)
Communicated by Editor's name

Abstract

This paper focuses on the resilient scheduling of parallel jobs on high-performance computing (HPC) platforms to minimize the overall completion time, or the makespan. We revisit the classical problem while assuming that jobs are subject to failures caused by transient or silent errors, and hence may need to be re-executed each time they fail to complete successfully. This work generalizes the classical framework where jobs are known offline and do not fail: in this framework, list scheduling that gives priority to the longest jobs is known to be a 3-approximation when imposing to use shelves, and a 2-approximation without this restriction. We show that when jobs can fail, using shelves can be arbitrarily bad, but unrestricted list scheduling remains a 2-approximation. The paper focuses on the design of several heuristics, some list-based and some shelf-based, along with different priority rules and backfilling strategies. We assess and compare their performance through an extensive set of simulations using both synthetic jobs and log traces from the Mira supercomputer.

Keywords: Resilience, scheduling, rigid parallel jobs, silent errors, list schedules, shelf schedules, approximation algorithms.

* A preliminary version of this paper [6] was published in the 22nd Workshop on Advances in Parallel and Distributed Computational Models (APDCM), May, 2020.

1 Introduction

One of the main challenges faced by today's HPC platforms is resilience, since such platforms are confronted with many failures or errors due to their large scale [34]. Indeed, the number of failures is known to grow proportionally with the number of nodes on a platform [24], and the largest supercomputers today experience several failures per day. There are two main classes of errors that can cause failures in an application's execution, namely, fail-stop and silent errors. While fail-stop errors cause the execution to terminate (e.g., due to hardware fault), large-scale platforms are also confronted with *silent errors*, or *silent data corruptions (SDCs)*. Such errors are caused by cosmic radiation or packaging pollution, striking either the cache or memory units (bit flips), or the CPU operations [38, 51]. Even though any bit can be corrupted, the execution continues (unlike fail-stop errors), hence the error is transient, but it may dramatically impact the result of a running application. Many silent errors can be accurately detected by verifying the data using dedicated, lightweight detectors (e.g., [25, 48, 11, 4, 22, 10]). In this work, we focus on job failures caused by silent errors, and we aim to design resilient scheduling heuristics while assuming the availability of ad-hoc detectors to detect such errors.

The problem of scheduling a set of independent jobs on parallel platforms with the goal of minimizing the total completion time, or the *makespan*, has been extensively studied (see Section 2). Jobs may be parallel and should be executed on a given number of processors for a certain duration; both the processor requirement and the execution time of each job are known at the beginning. Such jobs are called *rigid* jobs, contrarily to moldable or malleable jobs, whose processor allocations can vary at launch time or during execution [15]. While moldable or malleable jobs offer more flexibility in the execution, rigid jobs remain the most prevalent form of parallel jobs submitted on today's HPC systems, and we focus on rigid jobs in this paper.

Unlike the classical scheduling problem without job failures, we consider *failure-prone platforms*, where a job could fail due to silent errors. Hence, at the end of each job's execution, an SDC detector will flag if a silent error has occurred during its execution. In this case, the job must be re-executed until it has been successfully completed without errors. For a set of jobs, each execution may lead to a different failure scenario, depending upon the jobs that have experienced failures as well as the number of such failures. The objective is to minimize the makespan under any failure scenario, as well as the *expected makespan*, averaged over all possible failure scenarios, where each scenario is weighted by a probability that governs its occurrence under certain failure assumptions. Since a failure scenario is unknown a priori, the scheduling decisions must be made *dynamically* on-the-fly, whenever an error has been detected. As a result, even for the same set of jobs, different schedules may be produced, depending on the failure scenario that occurred in a particular execution.

Building upon the existing framework for scheduling parallel jobs without failures, we propose two scheduling strategies, namely, a *list-based* strategy and a *shelf-based* strategy. While list-based schedules have no restrictions on the starting times of the jobs, shelf-based schedules group all jobs into subsets of jobs having the same starting time (called shelves); a shelf of jobs can start its execution once the longest job from the previous shelf has completed. For list-based scheduling, practical systems also employ a combination of reservation and backfilling strategies with different job priority rules to increase the system utilization. On platforms with no failures, variants for all of these strategies exist that could achieve constant approximations for the makespan (see Section 2 for details). The main focus of this paper is to extend these existing heuristics to execution scenarios with job failures, and to experimentally compare their performance using a variety of job and platform configurations.

Our main contributions are the following:

- We propose a formal model for the problem of resilient scheduling of parallel jobs on failure-prone platforms. The model formulates the performance of an algorithm under both worst-case and expected executions.
- We design a resilient list-based strategy, and prove that its greedy variant is a $(2 - \frac{1}{P})$-approximation, and its reservation variant is a $(3 - \frac{4}{P+1})$-approximation, where P is the total number of processors. These results apply to both worst-case and expected makespans.
- We design a resilient shelf-based strategy, but we show that, under some failure scenarios, any

shelf-based algorithm has an unbounded approximation ratio, thus having a makespan that is arbitrarily higher than the optimal makespan in the worst case.
- We conduct an extensive set of simulations to evaluate and compare different variants of these heuristics using both synthetic jobs and log traces from the Mira supercomputer. The results show that the performance of these resilient scheduling heuristics is close to the optimal in practice, even when confronted with failures.

The rest of this paper is organized as follows: Section 2 describes the background of parallel job scheduling and presents some related work. The formal models and the problem statement are presented in Section 3. The key contributions of the paper are presented in Section 4, where we describe both list-based and shelf-based strategies, and analyze their performance. Section 5 presents an extensive set of simulation results and highlights the main findings. Finally, Section 6 concludes the paper and discusses future directions.

2 Background and Related Work

This section describes the background of scheduling rigid parallel jobs and reviews some related work. We start with a brief description of the different scheduling flavors and strategies in Section 2.1. In Section 2.2, we discuss the offline problem, where all jobs are known statically and available initially. Taking job failures into account calls for a dynamic schedule, because re-executions are decided on-the-fly after the completion of each job. We then review the online problem, where jobs are presented dynamically to the scheduler in Section 2.3. Our problem with job failures is harder than the offline problem, and is different from the online problem where jobs are submitted at arbitrary but fixed release times. Practical schedulers often use reservation and backfilling, and we review related work in this area in Section 2.4. Finally, with failures, job execution times are no longer deterministic, and we review scheduling strategies for stochastic jobs in Section 2.5.

2.1 Different Scheduling Flavors and Strategies

Historically, scheduling parallel jobs comes in two flavors: if a job requests p processors, either any subset of p processors can be assigned, or only subsets of p *contiguous* processors can be chosen. In the latter case, processors are organized as a linear array and labeled from 1 to P, where P is the total number of processors; then only neighboring processors (whose labels differ by one) can be assigned to a job. The *contiguous* variant is equivalent to the *rectangle strip packing* problem, where rectangles are to be stacked (without rotation) within a strip of width P: rectangle widths represent processor numbers, and rectangle heights represent execution times.

Most scheduling strategies also come in two flavors: either the schedule is restricted to building *shelves* (also referred to as *levels* in some literature), or it is unrestricted, in which case the jobs are often scheduled based on an ordered *list*. Shelves are subsets of jobs with the same starting time, and for which each of the P processors is used at most once: the height of a shelf is the length of its longest job; when the shorter jobs complete, their processors become idle, but these processors are not reassigned to other jobs until the completion of the longest job of the shelf. Thus, a shelf resembles a bookshelf, hence the name. Shelf-based schedules play an important role in HPC, because they correspond to batched execution scenarios, where jobs are grouped into batches that are scheduled one after another. Note that for shelf-based algorithms, the contiguous and non-contiguous variants collapse.

2.2 Offline Scheduling of Rigid Jobs

To minimize the makespan for a set of rigid jobs that are known statically and available initially (i.e., offline), the problem is obviously NP-complete, as it generalizes the problem of scheduling independent jobs on two processors, a variant of the 2-PARTITION problem [19]. Coffman et al. [12] showed that the Next-Fit Decreasing Height (NFDH) algorithm is 3-approximation, and the First-Fit Decreasing-Height (FFDH) algorithm is 2.7-approximation. Both algorithms are shelf-based. See the survey by Lodi et al. [33] for more results and lower bounds on the best possible

approximation ratio for shelf-based algorithms, and see Han et al. [23] for the intricate relationship between strip packing and bin packing.

For list-based scheduling, Baker et al. [3] showed that the Bottom-up Left-justified (BL) heuristic while ordering the jobs in decreasing processor requirement achieves 3-approximation. Turek et al. [44] showed that ordering jobs in decreasing execution time is also 3-approximation. Moreover, both algorithms guarantee contiguous processor allocations for all jobs. Without the contiguous processor constraint, several works [44, 18, 17] showed that the greedy list-scheduling heuristic achieves 2-approximation. Finally, Jansen [28] presented a $(3/2 + \epsilon)$-approximation algorithm for any fixed $\epsilon > 0$. This is the best result possible, since a lower bound on the approximation ratio is $3/2$, which holds even when considering asymptotic performance [29].

When jobs have precedence constraints among them, list scheduling is shown to be P-approximation in the worst case, which holds for many commonly used job-ordering rules [31, 16]. However, if jobs require no more than qP processors for any $0 < q < 1$, then the approximation ratio of greedy list scheduling is $\frac{(2-q)}{(1-q)P+1}$ [31, 16]. In our problem, a particular failure scenario can be regarded as a special case of the general precedence constraint, where each job forms a linear chain, but the failure instance is unknown to the scheduler beforehand.

2.3 Online Scheduling of Rigid Jobs

In an online problem, a set of rigid jobs arrive dynamically over time and information of a job is not known until the job has arrived. In this case, the greedy list-scheduling maintains a competitive ratio of 2 [36, 29]. Chen and Vestjens [9] showed a 1.3473 lower bound on the competitive ratio of any deterministic online algorithm even when all jobs are sequential. Shmoys et al. [40] showed that by collecting all jobs that arrive during a batch and then scheduling them together in the next batch, one can convert any c-approximation offline algorithm to a $2c$-competitive online algorithm.

In another online model referred to as One-By-One, jobs, although all arrive initially, are presented one at a time to the online scheduler and must be irrevocably scheduled before the next job can be revealed. Johannes [29] showed that greedy list-scheduling is P-competitive in the worst case, and presented a 12-competitive algorithm. Baker and Schwarz [2] extended the two shelf-based algorithms presented in [12] and showed that Next-Fit is 7.46-competitive and First-Fit is 6.99-competitive. The surveys by Csirik and Woeginger [14, 13] describe more results and lower bounds that use shelf-based algorithms in this model. The best known competitive ratio for One-By-One is 6.6623, obtained by Hurink and Paulus [26] and independently by Ye et al. [49].

The problem studied in this paper can be considered as semi-online, since all jobs are known to the scheduler initially but not their failure scenarios. We point out that the technique by Shmoys et al. [40] to obtain $2c$-competitiveness is not applicable here, because it relies on jobs having fixed, although unknown, release times, whereas the "new job arrival" times in our problem (corresponding to failed jobs restarting) depend on the decisions made on-the-fly by the schedulers.

2.4 Batch Schedulers in Practical Systems

In practical systems, parallel jobs are often scheduled by batch schedulers [27, 50, 43] that use a combination of *reservation* and *backfilling* strategies: while the high-priority jobs are scheduled by reserving processors in advance, the low-priority ones are used to fill in the "holes" to improve system utilization. Two popular backfilling strategies are *conservative* [35] and *aggressive* (a.k.a. *EASY*) [32, 41]. The former gives a reservation for every job in the queue and a lower-priority job is moved forward as long as it does not delay the reservation for any higher-priority job. The latter only gives reservation to the job at the head of the queue (i.e., the one with the highest priority) and backfilling is allowed without delaying this highest-priority job. Note that greedy list-scheduling can be considered as an even more aggressive strategy, where no job receives a reservation and all jobs are scheduled using backfilling. As jobs arrive over time, most practical schedulers use First-Come First-Serve (FCFS) in conjunction with these strategies to prevent job starvation, but no worst-case performance guarantee is known for such schedulers. Various priority rules have been evaluated to characterize and tune their performance for different performance metrics (see, e.g., [42, 20, 47]).

2.5 Scheduling Stochastic Jobs

When a job could fail during execution and has to be restarted, it can be regarded as a stochastic job, whose execution time depends on the number of failures. Most prior works on stochastic scheduling have considered *sequential* jobs whose execution times follow a known probability distribution. The book by Pinedo [39] and the survey by Niño-Mora [37] discuss many relevant results on stochastic scheduling. For offline problems (i.e., no new job arrival), the literature has focused on two models. In the *static* model, all scheduling decisions (i.e., job assignments to processors) are made beforehand, whereas in the *dynamic* model, scheduling decisions are made dynamically on the fly. While both models coincide when job execution times are deterministic, they lead to different results for stochastic jobs. Under the static model, Kleinberg et al. [30] showed an $O(1)$-approximation algorithm for jobs with arbitrary distributions. Goel and Indyk [21] obtained a 2-approximation for jobs with Poisson distribution and a PTAS for exponential distribution. Under the dynamic model, the *Longest Expected Processing Time* first (LEPT) algorithm is known to achieve the optimal expected makespan for jobs with exponential distributions [7, 46] or when all jobs follow a common distribution with a non-increasing hazard rate function [45]. For jobs with arbitrary distributions, a straightforward extension of the classical online list scheduling yields a 2-approximation [8].

In this paper, we adopt the dynamic stochastic scheduling model to handle parallel jobs with failures. However, there are two main differences: job execution times follow a discrete distribution, and a failure does not require the job to be immediately re-executed. We prove a 2-approximation for a greedy algorithm in terms of expected makespan, and experimentally evaluate several list-based and shelf-based heuristics with different priority rules and backfilling options.

3 Models

In this section, we formally present the models and the problem statement. We also state the main assumptions we make in this paper.

3.1 Job Model

We consider a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of n parallel jobs to be executed on a platform consisting of P identical processors. All jobs are released at the same time, corresponding to the batch scheduling scenario in an HPC environment. In this paper, we focus on *rigid* jobs, which must be executed with a fixed number of processors that is usually set by the user when the job is submitted[1]. For each job $J_j \in \mathcal{J}$, let $p_j \in \{1, 2, \ldots, P\}$ denote its fixed (integral) processor allocation, and let t_j denote its error-free execution time. The *area* of the job is defined as $a_j = p_j \cdot t_j$.

3.2 Failure Model

We consider job failures that manifest as *silent errors* or *silent data corruptions (SDCs)* [34] that could corrupt a job during execution. A silent error detector is assumed to be available for each job, which is triggered at the end of the job's execution. If an error is detected, the job needs to be re-executed, followed by another error detection. This process repeats until the job completes successfully without errors. Current state-of-the-art SDC detectors are typically lightweighted (e.g., ABFT for matrix computations [25, 48, 11], or data analytics for scientific applications [4, 22, 10]), and hence incur a negligible cost compared to the overall execution time of the job.

All the list-based and shelf-based scheduling heuristics introduced and compared in this paper are agnostic of the probability of each job to fail any given number of times. Specifically, for a job J_j, consider a particular run where it fails f_j times before succeeding on the $(f_j + 1)$-th execution. The probability that this happens is denoted as $q_j(f_j)$. Let $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ denote a *failure scenario*, i.e., a vector of the number of failed execution attempts for all jobs, during a particular run. Assuming that errors occur independently for different jobs, the probability that this combined

[1] Other parallel job models include *moldable* and *malleable* models, which allow a job's processor allocation to vary at launch time or during execution [15]. Considering alternative job models will be part of our future work.

failure scenario happens can be computed as $Q(\mathbf{f}) = \prod_{j=1\ldots n} q_j(f_j)$. The failure scenario $\mathbf{f}$, as well as the associated probabilities $q_j(f_j)$ and $Q(\mathbf{f})$ may be unknown to the scheduler.

3.3 Problem Statement

We study the following resilient scheduling problem: Given a set $\mathcal{J}$ of parallel jobs, find a schedule for $\mathcal{J}$ on P identical processors under any failure scenario $\mathbf{f} = (f_1, f_2, \ldots, f_n)$. Here, a *schedule* for $\mathbf{f}$ is defined by a collection $\mathbf{s} = (\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_n)$ of starting time vectors for all jobs, where vector $\vec{s}_j = (s_j^{(1)}, s_j^{(2)}, \ldots, s_j^{(f_j+1)})$ specifies the starting times for job J_j at different execution attempts until success.

The objective is to minimize the overall completion time of all jobs, or the *makespan*. Suppose an algorithm ALG makes scheduling decision $\mathbf{s}$ during the failure scenario $\mathbf{f}$, then the makespan of the algorithm for this scenario is defined as:

$$T_{\text{ALG}}(\mathbf{f}, \mathbf{s}) = \max_{j=1\ldots n} \left(s_j^{(f_j+1)} + t_j \right) . \tag{1}$$

All scheduling decisions should be made while satisfying the following two constraints:

- The number of processors utilized at any time t by the set $\mathcal{J}_t$ of running jobs should not exceed the total number P of available processors on the platform, i.e.,

$$\sum\nolimits_{J_j \in \mathcal{J}_t} p_j \leq P, \quad \forall t. \tag{2}$$

- A job cannot be re-executed if its previous execution attempt has not yet been completed, i.e.,

$$s_j^{(i+1)} \geq s_j^{(i)} + t_j, \quad \forall j = 1 \ldots n, \quad \forall i \geq 1. \tag{3}$$

This scheduling problem, encompassing the failure-free problem as a special case, is clearly NP-hard. A scheduling algorithm ALG is said to be a *c-approximation* if its makespan is at most c times that of an optimal scheduler for all possible sets of jobs, and for all possible failure scenarios, i.e.,

$$T_{\text{ALG}}(\mathbf{f}, \mathbf{s}) \leq c \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*), \tag{4}$$

where $T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*)$ denotes the optimal makespan with scheduling decision $\mathbf{s}^*$ under failure scenario $\mathbf{f}$. Clearly, this optimal makespan admits the following two lower bounds:

$$T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \geq t_{\max}(\mathbf{f}) , \tag{5}$$

$$T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \geq \frac{A(\mathbf{f})}{P} , \tag{6}$$

where $t_{\max}(\mathbf{f}) = \max_{j=1\ldots n}(f_j + 1) \cdot t_j$ is the maximum cumulative execution time of any job under $\mathbf{f}$, and $A(\mathbf{f}) = \sum_{j=1}^{n}(f_j + 1) \cdot a_j$ is the total cumulative area.

In Section 4, we establish several approximation results, which are valid for any failure scenario, regardless of its individual probability. This is the strongest result that can be obtained from a theoretical perspective. However, from a practical perspective, given a set of jobs, it is not easy to assess the performance of a scheduling heuristic if the probability $Q(\mathbf{f}) = \prod_{j=1\ldots n} q_j(f_j)$ of each failure scenario $\mathbf{f}$ is not known. Thus, for the experiments in Section 5, we report the expected cost of each heuristic under the standard exponential probability distribution, as explained below.

3.4 Expected Makespan

Suppose the occurrence of silent errors striking the jobs follows an exponential probability distribution, and that the mean time between error (MTBE) of an individual processor is μ, so the error rate of the processor is given by $\lambda = 1/\mu$. For a job J_j executed on p_j processors, the probability that the job is struck by a silent error during execution is then given by $q_j = 1 - e^{-\lambda p_j \cdot t_j} = 1 - e^{-\lambda a_j}$ [24]. Then, the probability for job J_i to fail f_j times before succeeding on the $(f_j + 1)$-th execution is $q_j(f_j) = q_j^{f_j}(1 - q_j)$.

Given a set $\mathcal{J}$ of parallel jobs, we can now define the *expected makespan* of an algorithm ALG, which is taken over all possible failure scenarios weighted by their probabilities, as:

$$\mathbb{E}(T_{\text{ALG}}) = \sum\nolimits_{\mathbf{f}} Q(\mathbf{f}) \cdot T_{\text{ALG}}(\mathbf{f}, \mathbf{s}) \ . \tag{7}$$

In this case, an algorithm is said to be a c-approximation if we have:

$$\mathbb{E}(T_{\text{ALG}}) \leq c \cdot \mathbb{E}(T_{\text{OPT}}) \ , \tag{8}$$

for all possible sets of jobs, where $\mathbb{E}(T_{\text{OPT}})$ denotes the optimal expected makespan. This is because the inequality is true for each failure scenario, hence for the weighted sum. Obviously, the converse is not true: an algorithm could satisfy Equation (8) (thus being a c-approximation in expectation) but be arbitrarily worse than the optimal on some (low probability) failure scenarios. Still, expected makespans provide a synthetic indicator on the performance of an algorithm under study and enable easy, quantitative comparisons. Thus, we use them for the experimental evaluations in Section 5.

3.5 Dynamic Scheduling

As all the information regarding the set of jobs is available, one approach to the problem is to make all scheduling decisions (i.e., starting times $\mathbf{s}$) *statically* at the beginning, and then execute the jobs according to this static schedule. While this approach works for failure-free jobs, it is problematic when jobs can fail and re-execute. In particular, since the failure scenario is not known in advance, a static schedule needs to pre-compute a long (possibly infinite) sequence of starting times for all jobs to account for every possible failure scenario, while ensuring the satisfaction of the scheduling constraints. Pre-computing such a static schedule would be computationally inefficient, especially when there turn out to be only a few failures in a particular run.

In contrast, another more flexible approach is to make scheduling decisions *dynamically* depending on the particular failure scenario that is unveiled from an execution. For example, a scheduling algorithm may decide the starting time for the next execution attempt of a job depending on the failure scenario and constructed schedule so far. As a result, even for the same set of jobs, the algorithm may produce different schedules in response to the different failure scenarios that could arise during runtime. In this paper, we adopt this dynamic approach.

4 Resilient Scheduling Heuristics

In this section, we present a resilient list-based heuristic (R-LIST) and a resilient shelf-based heuristic (R-SHELF) for scheduling rigid parallel jobs that could fail due to silent errors. We show that the greedy variant of R-LIST without reservations is a 2-approximation, and a variant with reservations is a 3-approximation with the LJF job priority rule. For R-SHELF, even though it provides a 3-approximation in the failure-free case, we show that any resilient shelf-based algorithm, regardless of the priority rule used, is $\Omega(\ln P)$-approximation in some failure scenario. We then propose an improved shelf-based heuristic (R-SHELFFILL) that could have better practical performance than R-SHELF. However, we show that even this improved heuristic is $\Omega(P)$-approximation when coupled with the LPT priority rule.

4.1 R-LIST Scheduling Heuristic

We present a resilient list-based scheduling algorithm, called R-LIST, that schedules any set of parallel jobs with the capability to handle failures. Algorithm 1 shows the pseudocode of R-LIST. It extends the classical batch scheduler that combines reservation and backfilling strategies. The algorithm first organizes all jobs in a list (or a queue) based on some priority rule. Then, whenever an existing job J_k completes and hence releases processors (at time 0, a virtual job J_0 can be considered to complete), the algorithm schedules the remaining jobs, if any, in the queue. First, it checks if job J_k completes with error. If so, the job will be inserted back into the queue, based on its priority,

Algorithm 1: R-List

Input: a set $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ of rigid jobs, with processor allocation p_j and error-free execution time t_j for each job $J_j \in \mathcal{J}$, a platform with P identical processors, and parameter m;
Output: a list-based schedule for all jobs in $\mathcal{J}$ till they complete successfully.

```
begin
    Insert all jobs into a queue Q according to some priority rule;
    whenever an existing job J_k completes do
        if error detected for J_k then
            Q.insert_with_priority(J_k);
        end
        if Q.is_empty() = false then
            // schedule high-priority jobs using reservation
            for j = 1, 2 ..., min(m, |Q|) do
                J_j ← Q(j);
                Give job J_j an earliest possible reservation without delaying the reservation of job
                  J_j', ∀j' = 1, ..., j − 1;
            end
            // schedule low-priority jobs using backfilling
            t ← get_current_time();
            for j = m + 1, ..., |Q| do
                J_j ← Q(j);
                if Job J_j can be scheduled at the current time t without delaying the reservation of
                  job J_j', ∀j' = 1 ... m then
                    start executing job J_j at the current time t;
                end
            end
        end
    end
end
```

to be rescheduled later. All jobs in the queue are divided into two groups: the first m jobs with the highest priorities are each given a reservation at the earliest possible time, provided that any reservation made should not delay the starting times of the higher-priority jobs; the subsequent jobs in the queue (if any) are then examined one by one and backfilled to start at the current time, if such backfilling does not affect any reservations for the higher-priority jobs.[2]

The R-List heuristic takes a parameter m, and depending on the value of m chosen, it resembles several different scheduling strategies known in theory and practice:

- $m = |Q|$ (Conservative backfilling [35]): this strategy makes reservations for all pending jobs in the queue;
- $m = 1$ (Aggressive or EASY backfilling [32, 41]): this strategy makes a reservation only for the job at the head of the queue, and uses backfilling to schedule all remaining jobs in the queue;
- $m = 0$ (Greedy scheduler [44, 18, 17]): this strategy does not make any reservation, and uses backfilling to schedule all jobs in the queue.

Note that, in the case of $m > 0$, and when a job J_k with high priority fails, it may be re-inserted back into the first part of the queue (i.e., among the top m jobs). This may require recomputing the existing reservations (made previously) for some jobs in the queue that have lower priority than J_k. From an analysis point of view, we can think of each job completion as a trigger, which deletes all previous reservations and makes a fresh round of reservation and backfilling decisions based on the updated queue.

In the following, we denote by Reservation this variant of R-List with reservations ($m > 0$), and by Greedy the variant with $m = 0$.

[2] For practical schedulers, this is typically implemented using two separate job queues, one for reservation and one for backfilling.

4.2 Approximation Ratios for R-List

We show that, under any failure scenario, Reservation with a particular priority rule is a $(3-\frac{4}{P+1})$-approximation, and that Greedy with any priority rule is a $(2-\frac{1}{P})$-approximation. According to Equation (8), these results directly imply the same approximation ratios for the respective heuristic variants in terms of the expected makespan.

To assist the analysis, we first define some notations below. Since R-List only allocates and de-allocates processors upon job completions (the starting time of a reservation is necessarily at a future job completion time as well), the entire schedule can be divided into a set of consecutive and non-overlapping intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_v\}$, where jobs only start (or complete) at the beginning (or end) of an interval, and v denotes the total number of intervals. Let $p(I_\ell)$ be the processor utilization (i.e., total number of allocated processors) during interval I_ℓ. As R-List never idles all processors unless all jobs complete successfully, we have $p(I_\ell) \geq 1$ for all $I_\ell \in \mathcal{I}$. Let $|I_\ell|$ denote the length of interval I_ℓ. The makespan of R-List under a particular failure $\mathbf{f}$ scenario can be expressed as $T(\mathbf{f}, \mathbf{s}) = \sum_{1 \leq \ell \leq v} |I_\ell|$.

4.2.1 Result for Reservation

We first consider the Reservation variant, and analyze its performance while applying a priority rule that favors large jobs and uses any priority for small jobs. We call this rule Ljf (Large Job First). Specifically, a job is said to be *large* if its processor allocation is at least $\frac{P+1}{2}$, and *small* otherwise. The Ljf rule specifies that: (1) all large jobs have higher priority than all small jobs; (2) the priorities for large jobs are based on decreasing processor allocation; and (3) the priorities for small jobs are defined arbitrarily.

The following proposition shows the performance of Reservation in any failure scenario using the above Ljf rule. The result matches the 3-approximation ratio [3, 44] known for failure-free jobs.

Proposition 1. *For any set of rigid parallel jobs under any failure scenario* $\mathbf{f}$*, the makespan of* Reservation *with the* Ljf *priority rule satisfies:*

$$T_{\mathrm{R}}(\mathbf{f}, \mathbf{s}) \leq (3 - \frac{4}{P+1}) \cdot T_{\mathrm{Opt}}(\mathbf{f}, \mathbf{s}^*) \ . \tag{9}$$

Proof. Let J_j be a last successfully completed job in the schedule. We divide the set $\mathcal{I} = \{I_1, I_2, \ldots, I_v\}$ of all intervals into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$, where $\mathcal{I}_1$ contains the intervals in which job J_j is executing (including all of its execution attempts), and $\mathcal{I}_2 = \mathcal{I} \backslash \mathcal{I}_1$. Let $T_1 = \sum_{I \in \mathcal{I}_1} |I|$ and $T_2 = \sum_{I \in \mathcal{I}_2} |I|$ denote the total lengths of all intervals in $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. Based on Equation (5), we have $T_1 = (f_j + 1) \cdot t_j(p_j) \leq t_{\max}(\mathbf{f}) \leq T_{\mathrm{Opt}}(\mathbf{f}, \mathbf{s}^*)$.

We will show that the processor utilization in any interval $I \in \mathcal{I}_2$ satisfies $p(I) \geq \frac{P+1}{2}$. First, we observe that all large jobs are completed sequentially (in decreasing order of processor allocation) at the beginning of the entire schedule, since no two large jobs can be scheduled at the same time, and no small (backfilling) jobs can delay their executions because large jobs have higher priority based on the Ljf rule. Thus, if an interval $I \in \mathcal{I}_2$ contains a large job, its processor allocation must satisfy $p(I) \geq \frac{P+1}{2}$.

Now, consider any interval $I \in \mathcal{I}_2$ after all the large jobs have completed, and suppose I lies in between the i-th execution attempt and the $(i+1)$-th execution attempt of J_j, where $0 \leq i \leq f_j$. Hence, if such an interval exists, it means that J_j is a small job (with $p_j \leq \frac{P+1}{2}$), as well as all remaining jobs that are to be executed. Let t be the time at the beginning of this interval I. Recall that we can consider Reservation to make a fresh round of reservations and backfillings based on the current job queue Q at time t. Let J_k be the first job in Q that cannot be scheduled (either reserved or backfilled) to run at t. We know that such a job always exists because of the $(i+1)$-th execution attempt of J_j, which is scheduled to run at a later time. Let $\mathcal{J}_t$ be the set of jobs already running at time t or just scheduled to run at time t before job J_k, and let $p(\mathcal{J}_t)$ be the total processor allocation of all jobs in $\mathcal{J}_t$. As J_k cannot be scheduled to run at time t, it must be due to $p(\mathcal{J}_t) + p_k \geq P + 1$. Since J_k is a small job, i.e., $p_k \leq \frac{P+1}{2}$, it implies that $p(I) \geq p(\mathcal{J}_t) \geq \frac{P+1}{2}$.

Thus, based on Equation (6) and since $p_j \geq 1$, we have $P \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \geq A(\mathbf{f}) \geq \frac{P+1}{2} \cdot T_2 + p_j \cdot T_1 \geq \frac{P+1}{2} \cdot T_2 + T_1$. The overall execution time of RESERVATION with the LJF priority rule therefore satisfies:

$$\begin{aligned} T_{\text{R}}(\mathbf{f}, \mathbf{s}) &= T_1 + T_2 \\ &\leq T_1 + 2 \cdot \frac{P \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) - T_1}{P+1} \\ &= \frac{2P}{P+1} \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) + \left(1 - \frac{2}{P+1}\right) \cdot T_1 \\ &\leq (3 - \frac{4}{P+1}) \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \ . \end{aligned}$$

□

4.2.2 Result for GREEDY

We now consider the GREEDY variant. The following proposition shows the performance of GREEDY in any failure scenario regardless of the priority rule. The result generalizes the same approximation ratio [44, 18, 17] of GREEDY for failure-free jobs.

Proposition 2. *For any set of rigid parallel jobs under any failure scenario* $\mathbf{f}$*, the makespan of* GREEDY *regardless of the priority rule satisfies:*

$$T_{\text{G}}(\mathbf{f}, \mathbf{s}) \leq (2 - \frac{1}{P}) \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \ . \tag{10}$$

Proof. Given the set $\mathcal{I} = \{I_1, I_2, \ldots, I_v\}$ of all intervals in the schedule, let $p_{\min} = \min_{1 \leq \ell \leq v} p(I_\ell)$ denote the minimum processor utilization among them. Since the algorithm never idles all processors unless all jobs complete successfully, we have $p_{\min} \geq 1$. We consider two cases:

Case 1: $p_{\min} \geq \frac{P+1}{2}$. In this case, we have $p(I_\ell) \geq p_{\min} \geq \frac{P+1}{2}$ for all $1 \leq \ell \leq v$. Hence, based on Equation (6), we get $P \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \geq A(\mathbf{f}) = \sum_{\ell=1,\ldots,v} |I_\ell| \cdot p(I_\ell) \geq \frac{P+1}{2} \cdot T_{\text{G}}(\mathbf{f}, \mathbf{s})$. This implies:

$$T_{\text{G}}(\mathbf{f}, \mathbf{s}) \leq \frac{2P}{P+1} \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \leq (2 - \frac{1}{P}) \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \ .$$

Case 2: $p_{\min} < \frac{P+1}{2}$. In this case, let $I_{\min}$ denote the last-executed interval that has processor utilization $p_{\min}$. Consider a job J_j that is running during interval $I_{\min}$. Necessarily, we have $p_j \leq p_{\min}$. We divide the set $\mathcal{I}$ of intervals into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$, where $\mathcal{I}_1$ contains the intervals in which job J_j is executing (including all of its execution attempts), and $\mathcal{I}_2 = \mathcal{I} \backslash \mathcal{I}_1$. Let $T_1 = \sum_{I \in \mathcal{I}_1} |I|$ and $T_2 = \sum_{I \in \mathcal{I}_2} |I|$ denote the total lengths of all intervals in $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. Based on Equation (5), we have $T_1 = (f_j + 1) \cdot t_j(p_j) \leq t_{\max}(\mathbf{f}) \leq T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*)$.

For any interval $I \in \mathcal{I}_2$ that lies between the i-th execution attempt and the $(i+1)$-th execution attempt of J_j in the schedule, where $0 \leq i \leq f_j$, the processor utilization of I must satisfy $p(I) \geq P - p_{\min} + 1$, since otherwise there are at least $p_{\min} \geq p_j$ available processors during interval I and hence the $(i+1)$-th execution attempt of J_j would have been scheduled at the beginning of I.

For any interval $I \in \mathcal{I}_2$ that lies after the (f_j+1)-th (last) execution attempt of J_j, there must be a job J_k running during I and that was not running during $I_{\min}$ (meaning no attempt of executing J_k was made during $I_{\min}$). This is because $p(I) > p_{\min}$, hence the job configuration must differ between I and $I_{\min}$. The processor utilization during interval I must also satisfy $p(I) \geq P - p_{\min} + 1$, since otherwise the processor allocation of J_k will be $p_k \leq p(I) \leq P - p_{\min}$, implying that the first execution attempt of J_k after interval $I_{\min}$ would have been scheduled at the beginning of $I_{\min}$.

Thus, for all $I \in \mathcal{I}_2$, we have $p(I) \geq P - p_{\min} + 1$. Based on Equation (6), we have $P \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \geq A(\mathbf{f}) \geq (P - p_{\min} + 1) \cdot T_2 + p_{\min} \cdot T_1$. Since $p_{\min} \geq 1$, the overall execution time of GREEDY therefore

satisfies:

$$\begin{aligned} T_{\mathrm{G}}(\mathbf{f},\mathbf{s}) &= T_1 + T_2 \\ &\le T_1 + \frac{P \cdot T_{\mathrm{OPT}}(\mathbf{f},\mathbf{s}^*) - p_{\min} \cdot T_1}{P - p_{\min} + 1} \\ &= \frac{P}{P - p_{\min} + 1} \cdot T_{\mathrm{OPT}}(\mathbf{f},\mathbf{s}^*) + \frac{P - 2p_{\min} + 1}{P - p_{\min} + 1} \cdot T_1 \\ &\le \frac{2P - 2p_{\min} + 1}{P - p_{\min} + 1} \cdot T_{\mathrm{OPT}}(\mathbf{f},\mathbf{s}^*) \\ &\le (2 - \frac{1}{P - p_{\min} + 1}) \cdot T_{\mathrm{OPT}}(\mathbf{f},\mathbf{s}^*) \\ &\le (2 - \frac{1}{P}) \cdot T_{\mathrm{OPT}}(\mathbf{f},\mathbf{s}^*) \ . \end{aligned}$$

□

4.3 R-Shelf Scheduling Heuristic

We now present a shelf-based scheduling heuristic, called R-Shelf, that schedules any set of parallel jobs onto a series of shelves while handling job failures.

Algorithm 2 shows the pseudocode of R-Shelf. As in R-List, the algorithm starts by organizing all jobs in a queue based on some priority rule. Whenever the jobs in the preceding shelf all complete (at time 0, a virtual shelf S_0 with no job on it can be considered to complete), the algorithm builds a new shelf and adds the remaining jobs to it. First, any job in the preceding shelf that completes with error will be inserted back into the queue based on its priority. Then, the algorithm scans the queue and adds a job to the new shelf if the job can fit in without violating the processor constraint. R-Shelf takes a binary parameter b that determines if backfilling is used in the process:

- $b = 0$ (No backfilling): the heuristic closes the new shelf upon encountering the first job in the queue that does not fit in the shelf. This resembles the Next-Fit (NF) strategy for bin-packing.
- $b = 1$ (Backfilling): the heuristic scans all the jobs in the queue until no more job can be added to the new shelf. This resembles the First-Fit (FF) strategy for bin-packing.

Once the jobs in the new shelf have been selected, they will simultaneously start their executions.

4.4 Lower Bounds for Shelf-Based Heuristics

For failure-free jobs, the variant of R-Shelf without backfilling and considering jobs in the non-increasing execution time order is equivalent to the Next-Fit Decreasing Height (NFDH) [12] algorithm for strip packing. The algorithm starts with the longest job J_1, which is put on the first shelf, whose height is t_1. Then, the next job J_2 is put on the same shelf if it fits in, meaning that $p_1 + p_2 \le P$, otherwise a new shelf is started for J_2, whose height is t_2. The algorithm proceeds like this, either putting the next job on the last shelf if it fits in, or creating a new shelf otherwise. Despite its simplicity, the algorithm is shown to be a 3-approximation for failure-free jobs [12, 44].

Now, when jobs can fail, we show that there exists a job instance $\mathcal{J}$ and a failure scenario $\mathbf{f}$ such that any shelf-based algorithm has a makespan $T_{\mathrm{S}}(\mathbf{f},\mathbf{s})$ that is arbitrarily higher than the optimal makespan $T_{\mathrm{OPT}}(\mathbf{f},\mathbf{s}^*)$ regardless of the job priority used. This is in clear contrast with the 3-approximation result for the failure-free case. Note that $T_{\mathrm{OPT}}(\mathbf{f},\mathbf{s}^*)$ is not necessarily the optimal makespan of a shelf-based schedule.

Proposition 3. *There exists a job instance and a failure scenario such that any shelf-based algorithm regardless of the job priority used has an approximation ratio of* $\Omega(\ln P)$.

Proof. Consider an instance with a set $\mathcal{J} = \{J_1, \ldots, J_P\}$ of P uniprocessor jobs, where $t_j = 1/j$ and $p_j = 1$ for $1 \le j \le P$. For the failure scenario $\mathbf{f}$, we let $f_j = j - 1$ for $1 \le j \le P$; hence, job J_1 does not fail, job J_2 fails once before success, and job J_P fails $f_P = P - 1$ times before success.

This instance is illustrated in Figure 1 for $P = 4$. Because the instance contains only P uniprocessor jobs, R-Shelf has no freedom at all: it schedules the first execution of all P jobs in the first shelf of height t_1, then the second execution of jobs J_2 to J_P in the second shelf of height t_2, and so

Algorithm 2: R-SHELF

Input: a set $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ of rigid jobs, with processor allocation p_j and error-free execution time t_j for each job $J_j \in \mathcal{J}$, a platform with P identical processors, and parameter b;
Output: a shelf-based schedule for all jobs in $\mathcal{J}$ till they complete successfully.

```
begin
    Insert all jobs into a queue Q according to some priority rule;
    i ← 0, S_i ← ∅, T_i ← 0;
    whenever an existing job J_k completes do
        t ← get_current_time();
        if error detected for J_k then
            Q.insert_with_priority(J_k);
        end
        if t = T_i and Q.is_empty() = false then
            i ← i + 1 and S_i ← ∅; // start a new shelf
            for j = 1, 2 ..., |Q| do
                J_j ← Q(j);
                if Job J_j can fit in shelf S_i then
                    S_i ← S_i ⋃ {J_j};
                else if b = 0 then
                    break ; // no backfilling
                end
            end
            T_i ← t + max_{J_j ∈ S_i} t_j;
            start executing all jobs in shelf S_i at the current time t;
        end
    end
end
```

on until the last shelf of height t_P, which includes only the P-th execution of job J_P. Therefore, the makespan of R-SHELF is $T_{\mathrm{S}}(\mathbf{f}, \mathbf{s}) = 1 + \frac{1}{2} + \cdots + \frac{1}{P} = \Theta(\ln P)$, while the optimal algorithm schedules the different executions of all jobs right after each other, thus having a makespan of $T_{\mathrm{OPT}}(\mathbf{f}, \mathbf{s}^*) = 1$.

Furthermore, since the P jobs have decreasing execution time and increasing number of failures, any shelf-based algorithm with any job priority will have at least one shelf of height t_i for all $1 \leq i \leq P$, thus having a makespan at least $T_{\mathrm{S}}(\mathbf{f}, \mathbf{s})$. Therefore, the same ratio applies to any shelf-based algorithm regardless of the job priority used. □

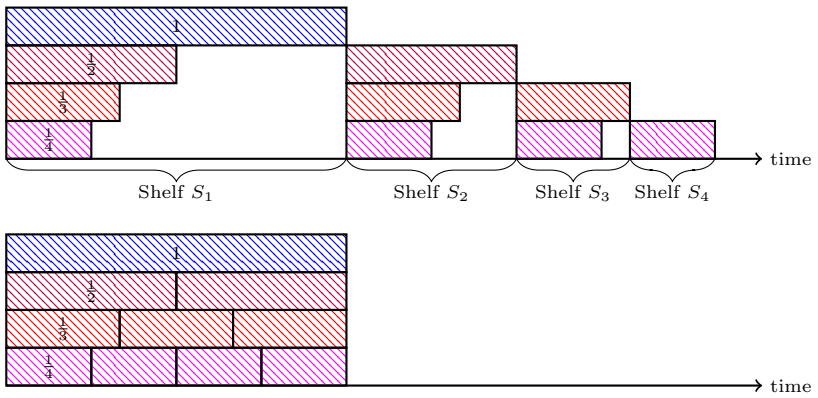


Figure 1: Illustration of the lower bound instance for $P = 4$: R-SHELF has a makespan of $\Theta(\ln P)$ (top), while the optimal algorithm is not shelf-based and has a makespan of 1 (bottom).

From the lower bound instance above, we can see that shelf-based heuristics may result in a lot of idle time, in particular because we wait until a shelf has completed before re-executing failed jobs. While keeping the simplicity of shelves, we consider a variant of the R-SHELF heuristic, called

Algorithm 3: R-ShelfFill

Input: a set $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ of rigid jobs, with processor allocation p_j and error-free execution time t_j for each job $J_j \in \mathcal{J}$, a platform with P identical processors, and parameter b;
Output: a shelf-filling-based schedule for all jobs in $\mathcal{J}$ till they complete successfully.
begin
  Insert all jobs into a queue Q according to some priority rule;
  $i \leftarrow 0, S_i \leftarrow \emptyset, T_i \leftarrow 0$;
  whenever an existing job J_k completes **do**
    $t \leftarrow get\_current\_time()$;
    if error detected for J_k **then**
      if $t + t_k \leq T_i$ **then**
        start re-executing J_k at the current time t; `// filling the shelf`
      else
        $Q.insert\_with\_priority(J_k)$;
      end
    end
    if $t = T_i$ and $Q.is\_empty() = false$ **then**
      $i \leftarrow i + 1$ and $S_i \leftarrow \emptyset$; `// start a new shelf`
      for $j = 1, 2 \ldots, |Q|$ **do**
        $J_j \leftarrow Q(j)$;
        if Job J_j can fit in shelf S_i **then**
          $S_i \leftarrow S_i \bigcup \{J_j\}$;
        else if $b = 0$ **then**
          break ; `// no backfilling`
        end
      end
      $T_i \leftarrow t + \max_{J_j \in S_i} t_j$;
      start executing all jobs in shelf S_i at the current time t;
    end
  end
end

R-ShelfFill, which keeps the structure of the shelves, but where failed jobs can be re-executed within the same shelf if they fit in, hence better *filling* the shelves. Specifically, let T_i denote the ending time of shelf S_i in the schedule (as defined by the maximum error-free execution time of all jobs placed onto the shelf). Then, we fill a shelf with re-executions of a failed job as long as they do not exceed T_i. Algorithm 3 shows the pseudocode of the R-ShelfFill heuristic, where the difference from the R-Shelf algorithm is highlighted (in red).

We focus here on the Lpt (Longest Processing Time) priority rule, which gives priorities to jobs with longer error-free execution times. Even though R-ShelfFill could improve upon R-Shelf in practice, we show that, with the Lpt priority rule, it still leads to a much longer makespan than the optimal with an approximation ratio of $\Omega(P)$, thus again defying the 3-approximation result known for the failure-free case when Lpt is used with the simple shelf-based algorithm.

Proposition 4. *There exists a job instance and a failure scenario such that* R-ShelfFill *with the* Lpt *priority rule has an approximation ratio of* $\Omega(P)$.

Proof. Consider the following instance, where all jobs are sequential (i.e., uniprocessor jobs) and classified into P different sets:

- The first set has one job with execution time 1, and $P-1$ jobs all with execution time $\frac{1+\epsilon}{P}$, where ϵ is arbitrarily close to 0. These first P jobs are not subject to failures.
- The second set has one job with execution time $1/P$ that fails $P-1$ times, so it has to be executed P times sequentially with a total execution time of 1. This set also contains $(P-1)P$ jobs that are not subject to failures, and they all have execution time $\frac{1+\epsilon}{P^2}$.
- In general, the i-th set (where $1 \leq i \leq P$) has one job with execution time $\frac{1}{P^{i-1}}$ that fails $P^{i-1}-1$ times, hence its P^{i-1} sequential executions cumulatively take time 1. Additionally, the i-th set contains $(P-1)P^{i-1}$ jobs with no failures, each with execution time $\frac{1+\epsilon}{P^i}$, which

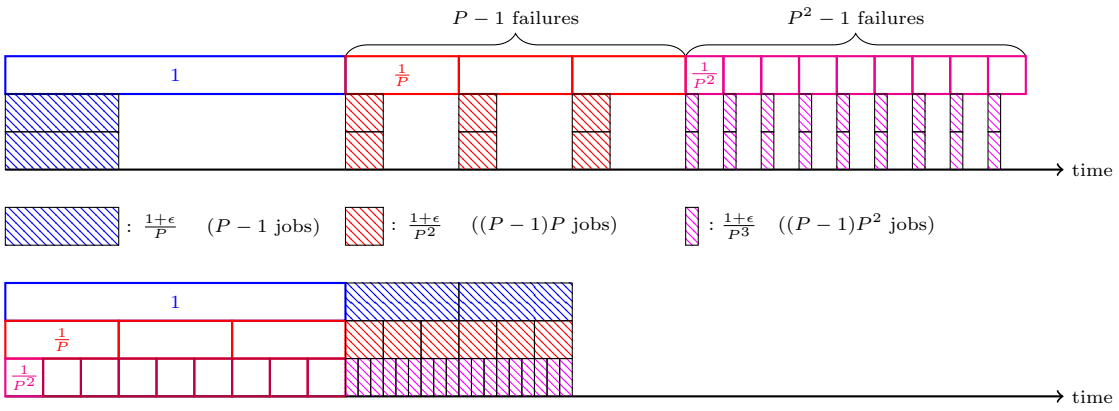


Figure 2: Illustration of the lower bound instance for $P = 3$: R-ShelfFill with Lpt priority has a makespan of P (top), while the optimal algorithm has a makespan no greater than 2 (bottom).

is slightly longer than the jobs from the next set.

This instance is illustrated in Figure 2 for $P = 3$. The R-ShelfFill heuristic with the Lpt priority rule will schedule jobs set by set. Since the small jobs in each set (*dashed* in the figure) are not subject to failures, R-ShelfFill is not able to fill more jobs on the same shelf, leading to a makespan of $T_{\mathrm{S}}(\mathbf{f}, \mathbf{s}) = P$. On the contrary, the optimal algorithm would first schedule the large jobs from all the sets together, completing them in a total duration of 1. Then, the remaining jobs can be executed with one set per processor and completed in time $(1 + \epsilon)\frac{P-1}{P} < 1$ for arbitrarily small ϵ. The optimal makespan therefore satisfies $T_{\mathrm{OPT}}(\mathbf{f}, \mathbf{s}^*) < 2$. □

We conclude this section with an open problem. Instead of a single failure scenario, consider an exponential probability distribution and the expected makespan as defined in Section 3.4. Will R-Shelf, R-ShelfFill, or any shelf-based algorithm admit a constant approximation ratio in expectation? To answer this question is difficult, because computing the expected makespan seems out of reach analytically. For the lower bound instance given in Proposition 3 with $P = 10$, we find numerically (using a computer program) that the expected makespan ratio of R-Shelf is 1.00005 for $\lambda = 10^{-7}$ and 1.07 for $\lambda = 10^{-3}$. We have not been able to construct an instance where this ratio (computed numerically) is greater than 3.

5 Performance Evaluation

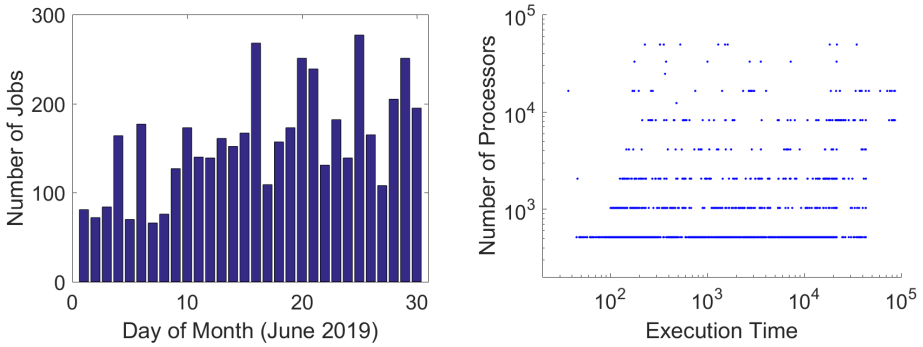
We now evaluate and compare the performance of all heuristics presented in Section 4, using different job priority rules and backfilling strategies. The evaluation is performed by simulation using both synthetic jobs and jobs extracted from the log traces of the Mira supercomputer.

5.1 Simulation Setup

We compare all proposed resilient scheduling heuristics combined with seven different job priority rules. The scheduling heuristics are:

- R-List-0: The list-based algorithm with $m = 0$;
- R-List-1: The list-based algorithm with $m = 1$;
- R-List-Q: The list-based algorithm with $m = |Q|$;
- R-Shelf-B: The R-Shelf algorithm with $b = 1$;
- R-Shelf-NB: The R-Shelf algorithm with $b = 0$;
- R-ShelfFill-B: The R-ShelfFill algorithm with $b = 1$;
- R-ShelfFill-NB: The R-ShelfFill algorithm with $b = 0$.

Note that, by construction, we expect R-ShelfFill-B to be more efficient than R-Shelf-B, and R-ShelfFill-NB to be more efficient than R-Shelf-NB. We first confirm this experimentally in Section 5.2, and then proceed by keeping only the better versions R-ShelfFill-B and R-ShelfFill-



(a) (b)

Figure 3: Data from the trace logs of the Mira supercomputer.

NB in the subsequent experiments together with R-List-0, R-List-1, and R-List-Q. Hence we compare only these five heuristics in Sections 5.3 and 5.4.

For each heuristic, we consider seven different job priority rules:

- Lpt/Spt (Longest/Shortest Processing Time): a job with a longer/shorter processing time will have higher priority;
- Hpa/Lpa (Highest/Lowest Processor Allocation): a job with a higher/lower number of requested processors will have higher priority;
- La/Sa (Largest/Smallest Area): a job with a larger/smaller area will have higher priority;
- Random: the priorities are determined randomly for all jobs.

We simulate two different settings, one using synthetic jobs and the other using real job traces from the Mira logs.

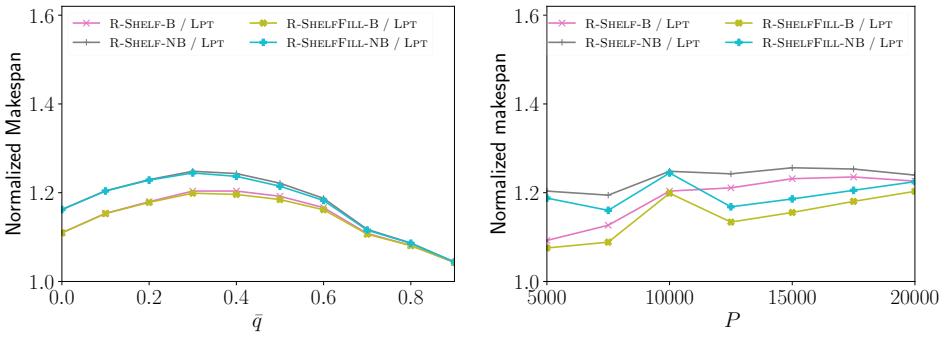
- *Synthetic jobs*: We generate 30 different job sets each with 100 jobs. For each job, the processor allocation is generated uniformly at random between 50 and 2000, while the execution time is generated uniformly at random between 100 and 20000 seconds. The total number of processors is set to be $P = 10000$. In the experiments, we also vary P to study its impact.
- *Jobs from Mira logs*: We generate jobs by extracting from the log traces [1] (of June 2019) of the Mira supercomputer, which has $P = 49152$ compute nodes. There were 4699 jobs submitted in June 2019, and we group the ones submitted each day as a set to form 30 sets of jobs. Figure 3(a) plots the number of jobs in each day of the month, varying between 66 and 277. The processor allocations of the jobs vary between 512 and 49152, and the execution times vary between 37 and 86494 seconds. Figure 3(b) plots these two parameters for all jobs in the month (with each point representing a job).

In both settings, silent errors are injected to the jobs based on the exponential distribution as described in Section 3.4. To study the impact of error rate, we further define the average failure probability for a set of jobs to be $\bar{q} = 1 - e^{-\lambda \bar{a}}$, where $\bar{a} = \sum_{j=1}^{n} a_j/n$ is the average area of all jobs in the set. Intuitively, $\bar{q}$ represents the probability that a job with the average area over all jobs would fail due to silent errors. For a given value of $\bar{q}$, we can compute the error rate as $\lambda = -\ln(1-\bar{q})/\bar{a}$, and hence the failure probability of any job J_j with area a_j to be

$$q_j = 1 - e^{-\lambda a_j} = 1 - (1-\bar{q})^{a_j/\bar{a}}$$

Based on this $\bar{q}$, we then randomly generate 1000 failure scenarios for the set of jobs following the probabilities q_j. For each failure scenario $\mathbf{f}$, we evaluate the makespans of the heuristics, normalized by the lower bound $L(\mathbf{f}) = \max(t_{\max}(\mathbf{f}), A(\mathbf{f})/P)$ as defined in Equations (5) and (6). The normalized makespans are then averaged over the 1000 failure scenarios for comparison.

The simulation code for all experiments is publicly available at `http://www.github.com/vlefevre/job-scheduling`.



(a) (b)

Figure 4: Normalized makespans of the different shelf-based heuristics with the Lpt priority over 30 sets of jobs when $\bar{q}$ varies between 0 and 0.9, and $P = 10000$ (left) and when P varies between 5000 and 20000, and $\bar{q} = 0.3$ (right).

5.2 Comparison of Shelf-Based Heuristics

We first compare the performance of the four shelf-based heuristics (R-Shelf-B, R-Shelf-NB, R-ShelfFill-B and R-ShelfFill-NB) using synthetic jobs. The goal is to assess how better the two R-ShelfFill variants perform with the Lpt priority rule, even though they have been shown to be not constant approximations. We study the impacts of two parameters, namely, the average failure probability $\bar{q}$ and the total number of processors P, on the performance of these heuristics. The results are averaged over the 30 different sets of jobs.

Figure 4(a) shows the impact of the failure probability $\bar{q}$ on the normalized makespans of the four heuristics while fixing $P = 10000$, and Figure 4(b) shows the impact of the total number of processors P while fixing $\bar{q} = 0.3$. The improvement of R-ShelfFill-B (resp. R-ShelfFill-NB) compared to R-Shelf-B (resp. R-Shelf-NB) seems small by looking at Figure 4(a), because $P = 10000$ is a particular value where both pairs of heuristics perform similarly. However, by looking at Figure 4(b), we observe that, when excluding the special value of $P = 10000$, R-ShelfFill-NB improves upon R-Shelf-NB by 4.9% on average and R-ShelfFill-B improves upon R-Shelf-B by 4.8% on average. Overall, the best shelf-based algorithm, R-ShelfFill-B, has a maximum normalized makespan of 1.2 in this set of experiments.

5.3 Comparison of All Heuristics

We now compare the performance of all five heuristics (excluding R-Shelf-B and R-Shelf-NB, since they are outperformed by the improved heuristics R-ShelfFill-B and R-ShelfFill-NB) using all job priority rules. Again, we use synthetic jobs, and focus on assessing the impacts of the average failure probability $\bar{q}$ and the total number of processors P. The results are averaged over the 30 different sets of jobs.

Figures 5(a)-(e) show the performance of the five heuristics with different job priorities when $\bar{q}$ varies from 0 to 0.9 while the number of processors is fixed at $P = 10000$. We can see that, for all list-based heuristics, the normalized makespans first increase with $\bar{q}$ and then decrease. Indeed, a higher failure probability will result in a larger number of errors, thus increasing the difficulty of scheduling and hence the makespan. However, when the probability is too high, an excessive number of errors will occur, making the optimal scheduler perform equally worse and thus reducing the makespan ratios for the heuristics. For the shelf-based heuristics, the normalized makespans decrease with $\bar{q}$ (except when using the Lpt priority). Here, jobs that fail need to wait for the completion of the current shelf to be re-executed, so the number of shelves is mainly determined by the number of re-executions, which influences both the makespan and an optimal scheduler when the failure probability is high. The normalized makespan is thus mainly decided by the efficiency

of the heuristic to fill one shelf. When the probability is small, the relative performance degrades because of the idle time induced by the shelves. We can also see that the LPT and LA priorities lead to the best performance for all list-based heuristics, with LPT performing better when $\bar{q}$ is low for R-LIST-1 and R-LIST-Q, and LA performing better for R-LIST-0 under any $\bar{q}$. For the shelf-based heuristics, LPT is the best priority, which is not surprising as the performance of these algorithms is mainly determined by the duration of each shelf.

Figure 5(f) further compares the performance of the five heuristics using the best priorities. While most list-based heuristics behave similarly when there is no failure (i.e., $\bar{q} = 0$), R-LIST-0 clearly outperforms the rest when jobs can fail. This corroborates the theoretical result that R-LIST-0 has the lowest approximation ratio regardless of the priority rule and failure scenario. Moreover, R-LIST-0 is also the heuristic that is least affected by job failures, with an increase in normalized makespan by less than 10% compared to the case of $\bar{q} = 0$, while the other heuristics experience 20-30% increase in normalized makespan. R-SHELFFILL-NB appears to be the worst heuristic for small failure probabilities with a makespan up to 18% higher than that of R-LIST-0 when $\bar{q} = 0.3$, while R-LIST-Q is the worst for medium and high probabilities with a makespan up to 26% higher than that of R-LIST-0 when $\bar{q} = 0.5$. The results are due to: (i) the restriction of R-SHELFFILL-NB for building shelves in a schedule, which leads to poor performance for some failure scenarios (such as the one discussed in Section 4.3), and hence an increase in the expected makespan, and (ii) the fact that R-LIST-Q is more affected by the increasing failure probability.

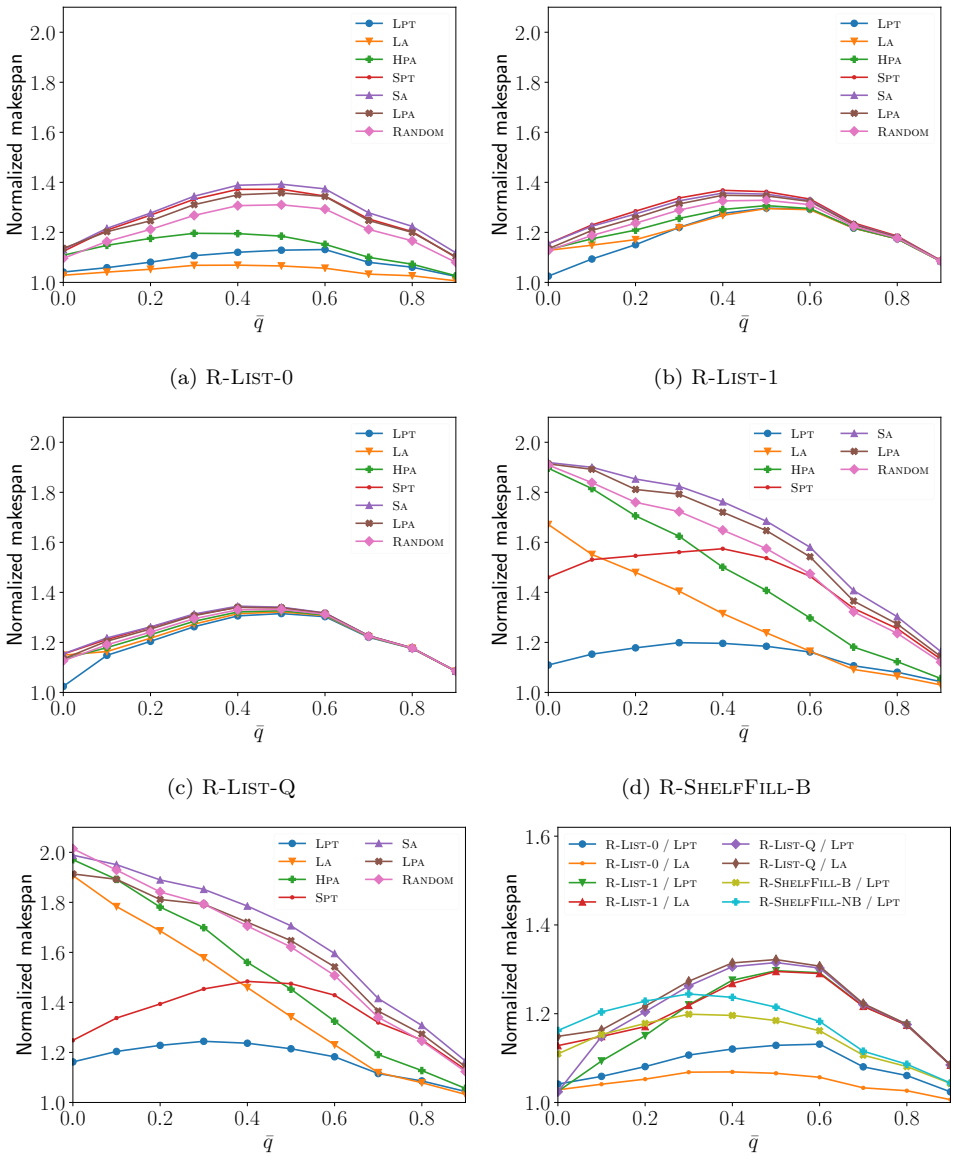
Figures 6(a)-(e) show the performance of the five heuristics with different job priorities when the number of processors P varies from 5000 to 20000 while the failure probability is fixed at $\bar{q} = 0.3$. Again, we can see that LA and LPT are the two best priority rules for all heuristics (except for R-SHELFFILL-NB where SPT is the second best), with LA performing better for R-LIST-0 and R-LIST-1, and LPT performing better for the other heuristics under all P. Also, the normalized makespans of the heuristics first increase with the number of processors and then tend to decrease. This is because when P is either too small (i.e., the total amount of resource is constrained) or too big (i.e., the total amount of resource is almost unconstrained), the optimal scheduler tends to have very similar performance as the heuristics.

Figure 6(f) further compares the performance of the five heuristics using some of the best priorities. As in the previous experiment, the best heuristic is R-LIST-0 with the LA priority, which is the least impacted by the total number of processors (with $<10\%$ variations in normalized makespan). Also, R-LIST-Q gives the worst performance (with a 23% increase in makespan compared to R-LIST-0 with LA when $P = 15000$) and has the largest variation ($\sim$20%) in normalized makespan as the number of processors changes.

From these experiments, we can see that job failures and processor variations do have an impact on the relative performance of different heuristics. Nevertheless, the makespans of all the heuristics (with good priorities) are never more than 40% worse than the theoretical lower bound, which can be much less than the optimal makespan. The results suggest the robustness of these heuristics, and that they should actually perform really well in practice, even with job failures.

5.4 Results Using Jobs from Mira

Finally, we evaluate the performance of different heuristics using real jobs from the Mira trace logs. Figures 7 and 8 show the normalized makespans of all heuristics and priority rules under all 30 days (sets) of jobs with and without failures. We observe that the LPT and LA priorities again offer the best performance, with LPT performing better this time for most job sets. This holds for every heuristic on average, especially when there is no failure (i.e., $\bar{q} = 0$). As the failure probability increases, both LPT and LA (and even HPA) give similar performance. The reason is that the processor allocations and execution times of the jobs in Mira are more skewed than those of the synthetic ones. Here, some jobs use a very large number of processors and have long execution times, which make them fail more often even with small values of $\bar{q}$. As a result, the makespan lower bound is largely determined by the total execution times of these jobs, thus any priority rule that favors these jobs will achieve similar performance. Comparing different heuristics, we can see that R-LIST-0 again performs the best and R-SHELFFILL-B the worse, especially with higher failure

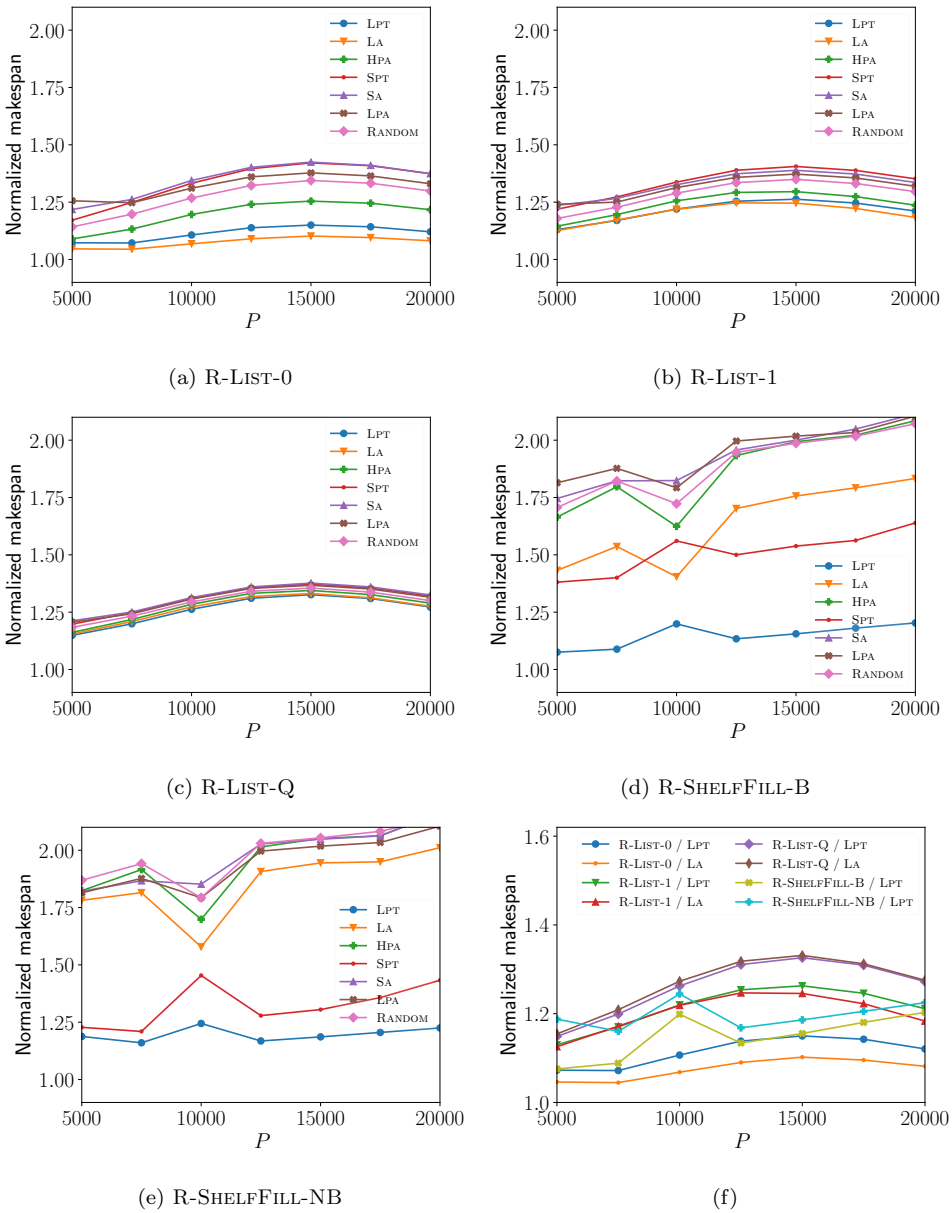


(a) R-List-0 (b) R-List-1

(c) R-List-Q (d) R-ShelfFill-B

(e) R-ShelfFill-NB (f)

Figure 5: Normalized makespans of different heuristics and priority rules over 30 sets of jobs when $\bar{q}$ varies between 0 and 0.9, and $P = 10000$.



(a) R-List-0 (b) R-List-1

(c) R-List-Q (d) R-ShelfFill-B

(e) R-ShelfFill-NB (f)

Figure 6: Normalized makespans of different heuristics and priority rules over 30 sets of jobs when P varies between 5000 and 20000, and $\bar{q} = 0.3$.

Table 1: Performance of different heuristics using LPT priority for all 30 days (sets) of jobs from June 2019 on the Mira supercomputer.

$\bar{q}$	Average #failures	Average makespan ratio					Standard deviation					Maximum makespan ratio				
		R-LIST			R-SHELFFILL		R-LIST			R-SHELFFILL		R-LIST			R-SHELFFILL	
		0	1	Q	B	NB	0	1	Q	B	NB	0	1	Q	B	NB
0	0	1.067	1.051	1.051	1.407	1.441	8.78×10^{-2}	8.19×10^{-2}	8.17×10^{-2}	1.27×10^{-1}	1.42×10^{-1}	1.425	1.425	1.425	1.633	1.760
0.05	15.2913	1.031	1.049	1.061	1.105	1.117	6.79×10^{-2}	6.87×10^{-2}	7.78×10^{-2}	1.20×10^{-1}	1.32×10^{-1}	1.278	1.292	1.292	1.475	1.495
0.1	254.453	1.016	1.025	1.028	1.052	1.054	4.66×10^{-2}	4.54×10^{-2}	4.97×10^{-2}	8.97×10^{-2}	9.28×10^{-2}	1.249	1.224	1.245	1.391	1.407

probability ($\bar{q} = 0.1$). This is consistent with the previous findings and corroborates the analysis.

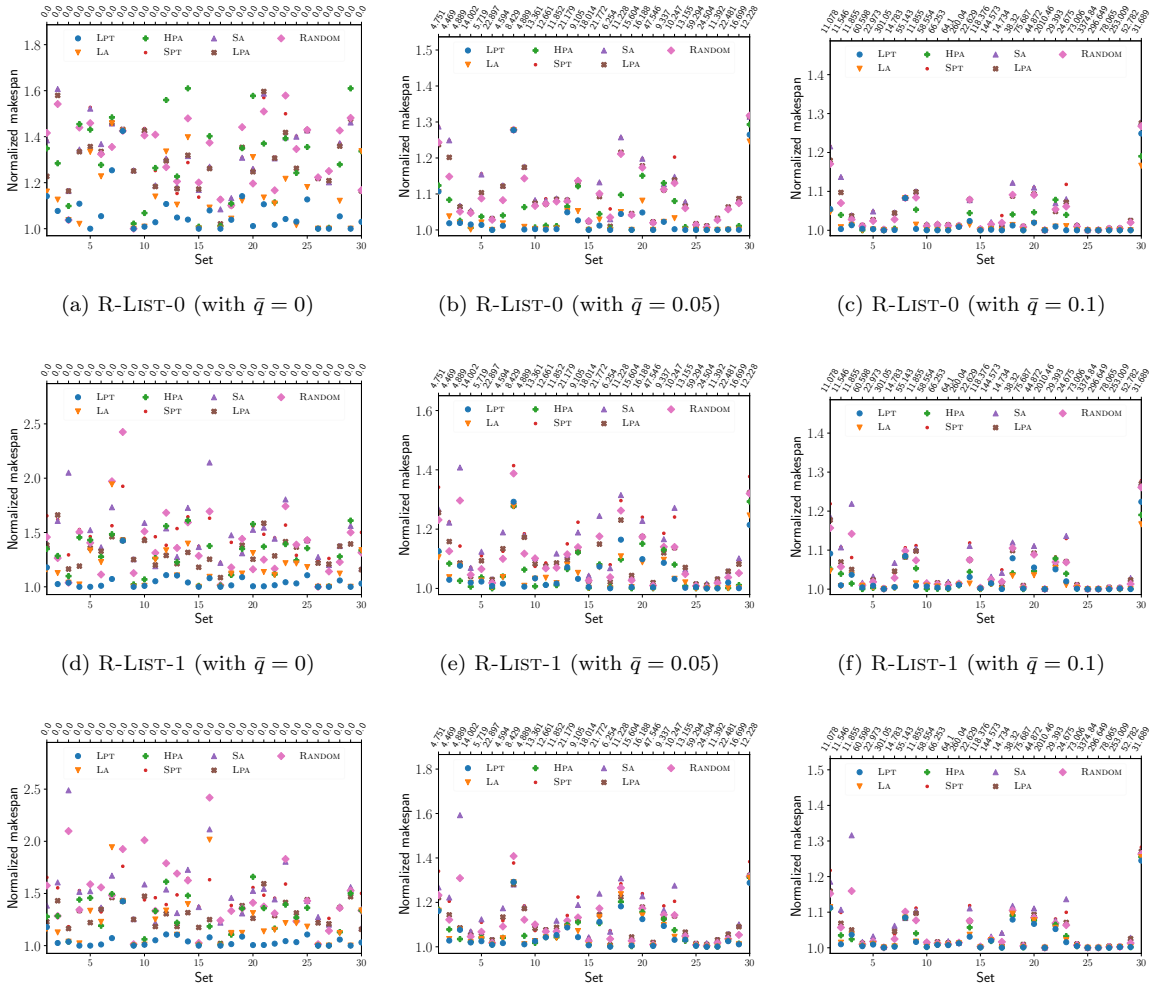
Table 1 summarizes the results of the five heuristics using the LPT priority (which is overall the best one) over 30 days (sets) of jobs, which have an average of 157.63 jobs per day (set). As $\bar{q}$ increases to 0.05 and 0.1, the average number of failures rises to around 15 and 254, respectively. All list-based heuristics have good average makespan ratios that are very close to 1 (with low standard deviations), as well as good maximum makespan ratios that are lower than 1.5, while the two shelf-based heuristics have worse performance in comparison, even when failures are not present. The maximum makespans, however, are never more than 80% of the theoretical lower bound. This again corroborates the results in Section 5.3.

Overall, these results confirm the efficacy and robustness of the resilient scheduling heuristics, not only for synthetic jobs, but also for real sets of jobs. In particular, both theory and practice have suggested that R-LIST-0 is the best heuristic when silent errors are present, and LPT and LA are the two best priorities for most cases. In all experiments we have conducted, this heuristic achieves a makespan that is within a few percent of the lower bound on average, and never more than 50% in the worst case.

6 Conclusion and Future Work

In this paper, we have investigated the problem of scheduling rigid parallel jobs onto an HPC platform subject to silent errors. We have revisited the classical scheduling algorithms in this new framework, where jobs that have been struck by errors must be re-executed (possibly many times) until success. We designed several resilient list-based and shelf-based scheduling heuristics, along with different priority rules and backfilling strategies. On the theoretical side, we proved that variants of the list-based heuristic achieve a constant approximation ratio (2 or 3 depending whether reservation is used or not). We also showed that any shelf-based heuristic is no longer a constant-factor approximation, while a failure-free variant was known to be a 3-approximation. Furthermore, we introduced a new variant of shelf-based heuristic that allows for re-executions of a failed job within the same shelf, provided that this does not increase the overall height of that shelf. Extensive simulations conducted using both synthetic jobs and real traces from the Mira supercomputer demonstrate that these heuristics are quite robust, and achieve makespans close to the optimal. As highlighted by the theoretical analysis, the best strategy remains the unrestricted greedy list-based scheduling with no reservations, and good results are obtained in practice when job priorities are assigned by processing times (favoring jobs with long execution times) or by areas (favoring jobs with many processors and/or long execution times).

Some problems remain open, in particular for the study of shelf-based algorithms, whose expected makespan under the exponential probability distribution is not known to be bounded by a constant factor of the optimal or not. A natural extension of this work would be to consider more flexible job models, such as moldable jobs (whose processor allocations can be decided at launch time) or malleable jobs (whose processor allocations can be changed during runtime). In [5], we have proved new approximation results for moldable jobs with several speedup profiles under any worst-case failure scenario. However, for jobs with arbitrary speedups, bounding the expected makespan in the average case remains an open question, since changing the number of processors assigned to a job may also change its failure probability, thereby severely complicating the problem. Another useful direction is to consider the impact of fail-stop errors on parallel jobs and design resilient scheduling algorithms to handle these errors.



(a) R-List-0 (with $\bar{q} = 0$)

(b) R-List-0 (with $\bar{q} = 0.05$)

(c) R-List-0 (with $\bar{q} = 0.1$)

(d) R-List-1 (with $\bar{q} = 0$)

(e) R-List-1 (with $\bar{q} = 0.05$)

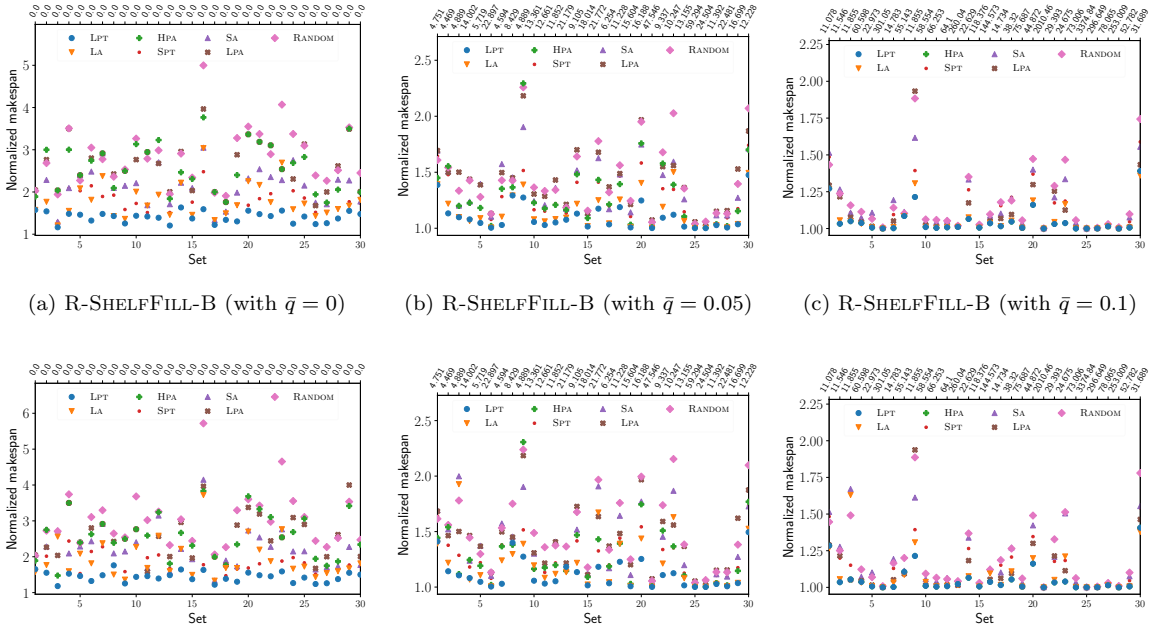
(f) R-List-1 (with $\bar{q} = 0.1$)

(g) R-List-Q (with $\bar{q} = 0$)

(h) R-List-Q (with $\bar{q} = 0.05$)

(i) R-List-Q (with $\bar{q} = 0.1$)

Figure 7: Performance of list-based heuristics for 30 job sets using the Mira trace logs (June 2019) with and without failures. Each row represents a different heuristic (R-List-0, R-List-1 and R-List-Q), and each column represents a different failure probability ($\bar{q} = 0$, $\bar{q} = 0.05$ and $\bar{q} = 0.1$). The average number of failures for each job set is indicated on top of each plot.



(a) R-ShelfFill-B (with $\bar{q} = 0$) (b) R-ShelfFill-B (with $\bar{q} = 0.05$) (c) R-ShelfFill-B (with $\bar{q} = 0.1$)

(d) R-ShelfFill-NB (with $\bar{q} = 0$) (e) R-ShelfFill-NB (with $\bar{q} = 0.05$) (f) R-ShelfFill-NB (with $\bar{q} = 0.1$)

Figure 8: Performance of shelf-based heuristics for 30 job sets using the Mira trace logs (June 2019) with and without failures. Each row represents a different heuristic (R-ShelfFill-B and R-ShelfFill-NB), and each column represents a different failure probability ($\bar{q} = 0$, $\bar{q} = 0.05$ and $\bar{q} = 0.1$). The average number of failures for each job set is indicated on top of each plot.

Acknowledgement

The data from Mira logs was generated from resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] Argonne Leadership Computing Facility. Mira log traces. https://reports.alcf.anl.gov/data/mira.html.

[2] B. Baker and J. Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12(3):508–525, 1983.

[3] Brenda S. Baker, E. G. Coffman, and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[4] Leonardo Bautista Gomez and Franck Cappello. Detecting silent data corruption through data dynamic monitoring for scientific applications. In *PPoPP*, 2014.

[5] Anne Benoit, Valentin Le Fèvre, Lucas Perotin, Padma Raghavan, Yves Robert, and Hongyang Sun. Scheduling moldable jobs on failure-prone platforms. Research Report RR-9340, INRIA, 2020.

[6] Anne Benoit, Valentin Le Fèvre, Padma Raghavan, Yves Robert, and Hongyang Sun. Design and comparison of resilient scheduling heuristics for parallel jobs. In *APDCM*, 2020.

[7] J. Bruno, P. Downey, and G. N. Frederickson. Sequencing tasks with exponential service times to minimize the expected flow time or makespan. *J. ACM*, 28(1):100–113, 1981.

[8] K. M. Chandy and P. F. Reynolds. Scheduling partially ordered tasks with probabilistic execution times. *SIGOPS Oper. Syst. Rev.*, 9(5):169–177, 1975.

[9] Bo Chen and Arjen P.A. Vestjens. Scheduling on identical machines: How good is LPT in an on-line setting. *Operations Research Letters*, 21(4):165–169, 1997.

[10] Chao Chen, Greg Eisenhauer, Matthew Wolf, and Santosh Pande. LADR: Low-cost application-level detector for reducing silent output corruptions. In *HPDC*, pages 156–167, 2018.

[11] Zizhong Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. *SIGPLAN Not.*, 48(8):167–176, 2013.

[12] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Comput.*, 9(4):808–826, 1980.

[13] János Csirik and Gerhard J. Woeginger. On-line packing and covering problems. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms: The State of the Art*, chapter 7, pages 147–177. Springer, 1998.

[14] János Csirik and Gerhard J. Woeginger. Shelf algorithms for on-line strip packing. *Information Processing Letters*, 63(4):171–175, 1997.

[15] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *JSSPP*, pages 1–34, 1997.

[16] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal on-line scheduling of parallel jobs with dependencies. *Journal of Combinatorial Optimization*, 1(4):393–411, 1998.

[17] Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, 130(1):49–72, 1994.

[18] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4(2):187–200, 1975.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[20] E. Gaussier, J. Lelong, V. Reis, and D. Trystram. Online tuning of EASY-backfilling using queue reordering policies. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2304–2316, 2018.

[21] Ashish Goel and Piotr Indyk. Stochastic load balancing and related problems. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, 1999.

[22] Pierre-Louis Guhur, Hong Zhang, Tom Peterka, Emil Constantinescu, and Franck Cappello. Lightweight and accurate silent data corruption detection in ordinary differential equation solvers. In *Euro-Par*, 2016.

[23] Xin Han, Kazuo Iwama, Deshi Ye, and Guochuan Zhang. Strip packing vs. bin packing. In Ming-Yang Kao and Xiang-Yang Li, editors, *Algorithmic Aspects in Information and Management*, pages 358–367. Springer, 2007.

[24] Thomas Herault and Yves Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag, 2015.

[25] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, 1984.

[26] Johann L. Hurink and Jacob Jan Paulus. Online algorithm for parallel job scheduling and strip packing. In Christos Kaklamanis and Martin Skutella, editors, *Approximation and Online Algorithms*, pages 67–74. Springer, 2008.

[27] David B. Jackson, Quinn Snell, and Mark J. Clement. Core Algorithms of the Maui Scheduler. In *JSSPP*, pages 87–102, 2001.

[28] Klaus Jansen. A $(3/2+\epsilon)$ approximation algorithm for scheduling moldable and non-moldable parallel tasks. In *SPAA*, pages 224–235, 2012.

[29] Berit Johannes. Scheduling parallel jobs to minimize the makespan. *J. of Scheduling*, 9(5):433–452, 2006.

[30] Jon Kleinberg, Yuval Rabani, and Éva Tardos. Allocating bandwidth for bursty connections. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 664–673, 1997.

[31] Keqin Li. Analysis of the list scheduling algorithm for precedence constrained parallel tasks. *Journal of Combinatorial Optimization*, 3(1):73–88, 1999.

[32] David A. Lifka. The ANL/IBM SP Scheduling System. In *JSSPP*, pages 295–303, 1995.

[33] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[34] Marc Snir et al. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, 2014.

[35] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[36] Edwin Naroska and Uwe Schwiegelshohn. On an on-line scheduling problem for parallel jobs. *Inf. Process. Lett.*, 81(6):297–304, 2002.

[37] José Niño-Mora. Stochastic scheduling. *Encyclopedia of Optimization*, pages 3818–3824, 2009.

[38] T.J. O'Gorman. The effect of cosmic rays on the soft error rate of a DRAM at ground level. *IEEE Trans. Electron Devices*, 41(4):553–557, 1994.

[39] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer-Verlag New York, Inc., Third edition, 2008.

[40] David B. Shmoys, Joel Wein, and David P. Williamson. Scheduling parallel machines on-line. *SIAM J. Comput.*, 24(6):1313–1331, 1995.

[41] Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. The EASY - LoadLeveler API Project. In *JSSPP*, pages 41–47, 1996.

[42] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *International Conference on Parallel Processing Workshop*, 2002.

[43] Garrick Staples. TORQUE resource manager. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[44] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate algorithms scheduling parallelizable tasks. In *SPAA*, 1992.

[45] R. R. Weber. Scheduling jobs with stochastic processing requirements on parallel machines to minimize makespan or flowtime. *J Appl Probab*, 19(1):167–182, 1982.

[46] G. Weiss and Pinedo M. Scheduling tasks with exponential service times on non-identical processors to minimize various cost functions. *J Appl Probab*, 17(1):187–202, 1980.

[47] Adam K. L. Wong and Andrzej M. Goscinski. Evaluating the EASY-backfill job scheduling of static workloads on clusters. In *CLUSTER*, 2007.

[48] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong Chen. Fault tolerant matrix-matrix multiplication: Correcting soft errors on-line. In *ScalA'11*, pages 25–28, 2011.

[49] Deshi Ye, Xin Han, and Guochuan Zhang. A note on online strip packing. *Journal of Combinatorial Optimization*, 17(4):417–423, 2009.

[50] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *JSSPP*, pages 44–60, 2003.

[51] J.F. Ziegler, M.E. Nelson, J.D. Shell, R.J. Peterson, C.J. Gelderloos, H.P. Muhlfeld, and C.J. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *IEEE Journal of Solid-State Circuits*, 33(2):246–252, 1998.