Resilient Scheduling of Moldable Parallel Jobs to Cope with Silent Errors

Anne Benoit, Valentin Le Fèvre, Lucas Perotin, Padma Raghavan, Yves Robert, Hongyang Sun

Abstract—We study the resilient scheduling of moldable parallel jobs on high-performance computing (HPC) platforms. Moldable jobs allow for choosing a processor allocation before execution, and their execution time obeys various speedup models. The objective is to minimize the overall completion time or the makespan, when jobs can fail due to silent errors and hence may need to be re-executed after each failure until successful completion. Our work generalizes the classical scheduling framework for failure-free jobs. To cope with silent errors, we introduce two resilient scheduling algorithms, LPA-LIST and BATCH-LIST, both of which use the LIST strategy to schedule the jobs. Without knowing a priori how many times each job will fail, LPA-LIST relies on a local strategy to allocate processors to the jobs, while BATCH-LIST schedules the jobs in batches and allows only a restricted number of failures per job in each batch. We prove approximation ratios for the two algorithms under several prominent speedup models (e.g., roofline, communication, Amdahl, power, monotonic, and a mix model). An extensive set of simulations is conducted to evaluate different variants of the two algorithms, and the results show that they consistently outperform some baseline heuristics. Overall, our best algorithm is within a factor of 1.6 of a lower bound on average over the entire set of experiments, and within a factor of 4.2 in the worst case.

Index Terms—Resilient scheduling, parallel jobs, moldable jobs, speedup model, failure scenario, transient errors, silent errors, list schedule, batch schedule, approximation ratios.

1 INTRODUCTION

Scheduling parallel jobs on high-performance computing (HPC) platforms is crucial for improving the application and system performance. In the scheduling literature, a *moldable* job is a parallel job that can be executed on an arbitrary but fixed number of processors, with an execution time depending on the number of processors on which it is executed. More precisely, a moldable job allows a variable set of resources for scheduling, but requires a fixed set of resources to execute. Hence, the job scheduler must allocate resources before starting the job. This corresponds to a variable static resource allocation, as opposed to a fixed static allocation (*rigid* jobs) and to a variable dynamic allocation (*malleable* jobs) [12]. Moldable jobs can easily adapt to the number of available resources, contrarily to rigid jobs, while being easy to design and implement, contrarily to malleable jobs. Thus, many computational kernels in scientific libraries are provided as moldable jobs that can be deployed on a wide range of processor numbers.

Because of the importance and wide availability of moldable jobs, scheduling algorithms for such jobs have been extensively studied. An important objective is to minimize the overall completion time, or makespan, for a set of jobs that are either all known before execution (offline setting) or released on-the-fly (online setting). Many prior works have published approximation algorithms or inapproximability results for both settings. These results notably depend upon the speedup model of the jobs. Indeed, consider a job whose execution time is $t(p)$ with p processors ($1 \leq p \leq P$, and P denotes the total number of processors on the platform). An arbitrary speedup model allows $t(p)$ to take any value, but realistic models call for $t(p)$ non-increasing with p: after all, if $t(p+1) > t(p)$, then why use that extra processor? Several speedup models have been introduced and analyzed, including the roofline model, the communication model, the Amdahl's model, the power model, and the (more general) monotonic model, where the area of the job $p \times t(p)$ is non-decreasing with p. Section 2 presents a survey of some important results for all these models.

In this paper, we revisit the problem of scheduling moldable jobs in a resilience framework. Unlike the classical problem without job failures, we consider *failure-prone jobs* that may need to be re-executed several times before successful completion. This is primarily motivated by the threat of silent errors (a.k.a. *silent data corruptions or SDCs*), which strike large-scale high-performance computing (HPC) platforms at a rate proportional to the number of floating-point (CPU) operations and/or the memory footprint of the applications (bit flips) [32], [41]. When a silent error strikes, 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. Coping with silent errors is a major challenge on today's HPC platforms [28] and it will become even more important at exascale [17]. Fortunately, many silent errors can be accurately detected by verifying the integrity of data using dedicated, lightweight detectors (e.g., [7], [15], [38]). When considering job failures caused by silent errors, we assume the availability of ad-hoc detectors.

To model this resilient scheduling problem, we focus

- *A preliminary version of this work has been published in the proceedings of the IEEE Cluster'20 conference [4]. Anne Benoit, Lucas Perotin and Yves Robert are with the LIP laboratory at Ecole Normale Supérieure de Lyon, France. Yves Robert is also with University of Tennessee Knoxville, USA. Valentin Le Fèvre is with Barcelona Supercomputing Center, Spain. Padma Raghavan is with Vanderbilt University, USA. Hongyang Sun is with the University of Kansas, USA. Contact: anne.benoit@ens-lyon.fr* on a general setting, where the aim is to schedule a set of moldable jobs subject to a failure scenario that specifies the number of failures for each job before successful completion. The failure scenario is, however, not known a priori, but only discovered as failed executions manifest themselves when the jobs complete. Hence, the scheduling decisions must be made *dynamically* on-the-fly: whenever an error has been detected, the job must be re-executed. 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. Intuitively, the problem lies in between an offline problem (where all the jobs are known before the execution starts) and an online problem (where the jobs are revealed on-the-fly). The goal is to minimize the makespan for any set of jobs under any failure scenario. Since the problem is $\mathcal{NP}$-complete (as it generalizes the $\mathcal{NP}$-complete failure-free scheduling problem), we aim at designing approximation algorithms that guarantee a makespan within a provable factor of the optimal makespan, independently of the jobs' failure scenarios.

Extending the literature on scheduling moldable jobs in the failure-free setting, this work lays the theoretical and practical foundation for scheduling such jobs on failure-prone platforms. Our key contributions are the design and analysis of two resilient scheduling algorithms with new approximation results for various speedup models. We further show that the two algorithms achieve good practical performance using an extensive set of simulations. The following summarizes our main results:

- We present a formal model for the problem of resilient scheduling of moldable jobs on failure-prone platforms. The model formulates both the worst-case and average-case performance of an algorithm for general speedup models and under arbitrary failure scenarios.
- We design a resilient scheduling algorithm, called LPA-LIST, that relies on a local processor allocation strategy and list scheduling to achieve $O(1)$-approximation for some prominent speedup models, including the roofline model, the communication model, the Amdahl's model, and a mix model. For the communication model, our approximation ratio improves on that of the literature for failure-free jobs. We also show that the algorithm is $\Theta(P^{1/4})$-approximation for the power model and $\Theta(P^{1/2})$-approximation for the general monotonic model. All of these results apply to both worst-case and average-case performance.
- We design another resilient scheduling algorithm, called BATCH-LIST, which schedules the jobs in batches using the list strategy, and each job is allowed only a restricted number of failures per batch. We prove a tight $\Theta(\log_2 f_{\max})$-approximation for the algorithm under arbitrary speedup model in the worst case, where $f_{\max}$ is the maximum number of failures of any job in a failure scenario. We also prove an $\omega(1)$ lower bound on the average-case performance of the algorithm.
- We conduct an extensive set of simulations to evaluate and compare different variants of the two algorithms. The results show that they consistently outperform some baseline heuristics. In particular, the first algorithm (LPA-LIST) performs better for the roofline and communication models, while the second algorithm (BATCH-LIST) performs better for the other models. Overall, our best algorithm is within a factor of 1.6 of a lower bound on average and within a factor of 4.2 in the worst case for all speedup models.

The rest of this paper is organized as follows. Section 2 surveys related work. The formal model and problem statement are presented in Section 3. In Section 4, we describe the two main algorithms and analyze their performance, providing several new approximation results. 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 RELATED WORK

We first review related work for **offline scheduling of independent moldable jobs** in the failure-free setting. All jobs are known a priori along with each job's execution time $t(p)$ as a function of the processor allocation p.

With the communication model, assuming a communication overhead when using more than one processor, Havill and Mao [16] presented a shortest execution time (SET) algorithm, which selects a number of processors that minimizes the job's execution time (they use around $\sqrt{w/c}$ processors when $t(p) = w/p + (p-1)c$), and schedules each job as early as possible. They showed that SET has an approximation ratio around 4. In this paper, we present an improved algorithm with an approximation ratio of 3. Furthermore, the algorithm is able to handle job failures. Dutton and Mao [11] presented an earliest completion time (ECT) algorithm, which allocates processors for each job that minimizes its completion time based on the current schedule. They proved tight approximation ratios of ECT for $P \leq 4$ processors and presented a general lower bound of 2.3 for arbitrary P. Kell and Havill [24] presented algorithms with improved approximation ratios for $P \leq 3$ processors.

The monotonic model assumes that the execution time is a non-increasing function and the area (product of processor allocation and execution time) is a non-decreasing function of the processor allocation. Examples of this model include Amdahl's speedup [1], i.e., $t(p) = w(\frac{1-\gamma}{p} + \gamma)$ with $\gamma \in [0,1]$, and the power speedup $t(p) = w/p^{\delta}$ [14], [35] with $\delta \in [0,1]$. Belkhale and Banerjee [2] presented a $2/(1+1/P)$-approximation algorithm by starting from a sequential LPT schedule and then iteratively incrementing the processor allocations. Błażewicz et al. [6] presented a 2-approximation algorithm while relying on an optimal continuous schedule, in which the processor allocation of a job may not be integral. Mounié et al. [30] presented a $(\sqrt{3}+\epsilon)$-approximation algorithm using a two-phase approach and dual approximation. Using the same techniques, they later improved the approximation ratio to $1.5+\epsilon$ [31]. Jansen and Land [20] showed the same $1.5+\epsilon$ ratio in special cases and proposed a PTAS.

In the arbitrary model, the execution time $t(p)$ is an unrestricted function of the processor allocation p. This model can be reduced to the monotonic model by scanning all possible allocations and discarding those with both larger execution time and area. Turek et al. [36] presented a 2-approximation list-based algorithm and a 3-approximation shelf-based algorithm. Ludwig and Tiwari [27] improved the 2-approximation result with lower runtime complexity. When each job only admits a subset of all possible processor allocations, Jansen [19] presented a $(1.5+\epsilon)$-approximation algorithm, which is the strongest result possible for any polynomial-time algorithm, since the problem does not admit an approximation ratio better than 1.5 unless $\mathcal{P}=\mathcal{NP}$ [23]. However, when the number of processors is a constant or polynomially bounded by the number of jobs, Jansen et al. [21] showed that a PTAS exists.

We now review work on **online scheduling**, where jobs are released one by one to the scheduler, and each released job must be scheduled irrevocably before the next job is revealed. As some algorithms discussed in the offline case (e.g., [11], [16], [24]) make scheduling decisions independently for each job, their results can be directly applied to this online problem with the corresponding competitive ratios. In contrast, other algorithms rely on information about all jobs to make global scheduling decisions, so these algorithms and their approximation results are not directly applicable to the online problem. In this online problem under the arbitrary speedup model, Ye et al. [39] presented a technique to transform any ρ-bounded algorithm[1] for rigid jobs to a 4ρ-competitive algorithm for moldable jobs. Then, relying on a 6.66-bounded algorithm for rigid jobs [18], [40], they gave a 26.65-competitive algorithm for moldable jobs. Both algorithms are based on building shelves. They also provided an improved 16.74-competitive algorithm [39].

The problem studied in this paper can be considered as semi-online, since all jobs are known to the scheduler offline but their failure scenarios are revealed online. We point out that the transformation technique by Ye et al. [39] does not apply here, since it implicitly assumes the independence of all jobs, whereas the different executions of the same job in our problem (due to failures) have linear dependence.

Finally, we discuss the problem of scheduling moldable jobs **with dependencies** modeled as directed acyclic graphs (DAGs). Under the roofline model, Wang and Cheng [37] showed that the earliest completion time (ECT) algorithm is a $(3-2/P)$-approximation. Feldmann et al. [13] proposed an online algorithm that maintains a system utilization at least α for some $\alpha \in (0,1]$. By choosing α carefully, they showed that the algorithm achieves 2.618-competitiveness, even when the job execution times and the DAG structure are unknown. Under the monotonic model, Belkhale and Banerjee [3] presented a 2.618-approximation algorithm while relying on the availability of an optimal processor allocation strategy to minimize the maximum of critical path length and total area. When assuming that the area of a job is a concave function of the number of processors, Jensen and Zhang [22] proposed a 3.29-approximation algorithm via a linear programming formulation. Chen and Chu [8] improved the ratio to around 2.95 by further assuming that the execution time of a job is strictly decreasing in the number of allocated processors.

For the problem studied in this paper, the jobs can be considered to form multiple linear chains, where each chain represents a job and the number of nodes in a chain represents the number of executions for the job. However, the failure scenario (thus the complete graph) is not known a priori, which prevents the above algorithms (except the ones in [13], [37]) from being directly applicable, since they all rely on knowing the complete graph in advance.

3 Models

In this section, we formally describe the models, and present the resilient scheduling problem.

3.1 Job and Speedup Models

We consider a set $\mathcal{J}=\{J_1,J_2,\dots,J_n\}$ 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. We focus on *moldable* jobs, which can be executed using any number of processors at launch time. The number of processors allocated cannot be changed once a job has started executing. For each job $J_j \in \mathcal{J}$, $t_j(p_j)$ denotes its execution time when allocated $p_j \in \{1,2,\dots,P\}$ processors[2], and the *area* of the job is defined as $a_j(p_j)=p_j \times t_j(p_j)$.

Let w_j denote the total work of job J_j (or its sequential execution time $t_j(1)$). The parallel execution time $t_j(p_j)$ of the job when allocated p_j processors depends on the speedup model. We consider several speedup models:

- *Roofline model*: linear speedup up to a bounded degree of parallelism $\bar{p}_j \in [1,P]$, i.e., $t_j(p_j)=w_j/p_j$ for $p_j \le \bar{p}_j$, and $t_j(p_j)=w_j/\bar{p}_j$ for $p_j > \bar{p}_j$;
- *Communication model*: there is a communication overhead $c_j \ge 0$ per processor when more than one processor is used, i.e., $t_j(p_j)=w_j/p_j+(p_j-1)c_j$;
- *Monotonic model*: the execution time (resp. area) is a non-increasing (resp. non-decreasing) function of the number of allocated processors, i.e., $t_j(p_j) \ge t_j(p_j+1)$ and $a_j(p_j) \le a_j(p_j+1)$;
- *Amdahl's model*: this is a particular case of the monotonic model with $t_j(p_j)=w_j\left(\frac{1-\gamma_j}{p_j}+\gamma_j\right)$, where $\gamma_j \in [0,1]$ denotes the inherently sequential fraction of the job;
- *Mix model*: this model combines Roofline, Communication and Amdahl's models with $t_j(p_j)=\frac{w_j(1-\gamma_j)}{\min(p,\bar{p}_j)}+w_j\gamma_j+(p_j-1)c_j$, which could capture more realistically the speedups of some complex applications;
- *Power model*: this is another particular case of the monotonic model with $t_j(p_j)=w_j/p_j^{\delta_j}$, where $\delta_j \in [0,1]$ is a constant parameter;
- *Arbitrary model*: there are no constraints on $t_j(p_j)$.

In all of these models, the *speedup* of job J_j with p_j processors is given by $\sigma_j(p_j)=\frac{t_j(1)}{t_j(p_j)}$.

3.2 Failure Model

We consider silent errors (or SDCs) that could cause a job to produce erroneous results after an execution attempt. Fortunately, such errors can often be detected using lightweight detectors (e.g., [7], [15], [38]) at the end of the job execution. The overhead of running these detectors is typically low and can also be reduced from parallel execution. Throughout this paper, we assume that the execution time of the job includes the cost of running a detector. If errors are detected,

1. An algorithm for rigid jobs is ρ-bounded if its makespan is at most ρ times the lower bound $L=\max\left(\frac{\sum_j t_j p_j}{P}, \max_j t_j\right)$, where t_j is the execution time of job J_j, and p_j is its processor allocation.

2. In this work, we do not allow fractional processor allocation, which could otherwise be implemented by time-sharing a processor among multiple jobs.

the job needs to be re-executed followed by another error detection. This process repeats until the job completes successfully without errors.

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 execution of the job set $\mathcal{J}$. Note that the number of times a job will fail is unknown to the scheduler a priori, and the failure scenario $\mathbf{f}$ becomes known only after all jobs have successfully completed without errors.

3.3 Problem Statement

We study the following *resilient scheduling* problem: Given a set of n moldable jobs, find a schedule on P identical processors under any failure scenario $\mathbf{f}$. In this context, a *schedule* is defined by the following two decisions:

- *Processor allocation*: a collection $\mathbf{p} = (\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n)$ of processor allocation vectors for all jobs, where vector $\vec{p}_j = (p_j^{(1)}, p_j^{(2)}, \ldots, p_j^{(f_j+1)})$ specifies the number of processors allocated to job J_j at different execution attempts until success. Note that processor allocation can change for each new execution attempt of a job.
- *Starting time*: 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 *makespan*, under any failure scenario. Suppose an algorithm makes decisions $\mathbf{p}$ and $\mathbf{s}$ for a job set $\mathcal{J}$ during a failure scenario $\mathbf{f}$. Then, the makespan of the algorithm for this scenario is defined as:

$$T(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) = \max_{1 \le j \le n} \left(s_j^{(f_j+1)} + t_j(p_j^{(f_j+1)}) \right) . \quad (1)$$

Both scheduling decisions should be made with the following two constraints: (1) the number of processors used at any time should not exceed the total number P of available processors; (2) a job cannot be re-executed if its previous execution attempt has not yet been completed.

As the problem generalizes the failure-free moldable job scheduling problem, which is known to be $\mathcal{NP}$-complete for $P \ge 5$ processors [10], the resilient scheduling problem is also $\mathcal{NP}$-complete. We therefore consider approximation algorithms. A scheduling algorithm ALG is said to be an *r-approximation*[3] if its makespan is at most r times that of an optimal scheduler for any job set $\mathcal{J}$ under any failure scenario $\mathbf{f}$, i.e.,

$$\sup_{\mathcal{J}, \mathbf{f}} \frac{T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s})}{T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)} = r , \quad (2)$$

where $T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)$ denotes the makespan produced by an optimal scheduler with scheduling decisions $\mathbf{p}^*$ and $\mathbf{s}^*$.

3.4 Worst-Case vs. Average-Case Analysis

The problem above is agnostic of the failure scenario, which is given as an input of the scheduling problem. A scheduling algorithm is an r-approximation only if it achieves a makespan at most r times the optimal for *any possible* failure scenario. This can be viewed as the *worst-case* analysis.

3. We consider the studied problem *offline*, although the failure scenario is unknown to the scheduler a prior and only revealed on-the-fly as jobs complete. One can also view the problem as *semi-online*, in which case all of our obtained approximation ratios can be interpreted as competitive ratios.

In contrast, some practical settings may call for an *average-case* analysis. In practice, each job $J_j \in \mathcal{J}$ could fail with a probability q_j in each execution attempt, independent of the number of previous failures. For instance, consider silent errors that strike CPUs and registers during the execution of a job: the probability of having a silent error is determined solely by the number of flops of the job, or equivalently, by its sequential execution time. On the contrary, the number of processors used to execute the job does not matter, even if the parallel execution time depends on the number of allocated processors. Suppose the occurrence of silent errors follows an exponential distribution with rate λ, then the failure probability for job J_j is given by:

$$q_j = 1 - e^{-\lambda t_j(1)} , \quad (3)$$

where $t_j(1)$ denotes the sequential execution time of job J_j. Then, the probability that the job fails f_j times before succeeding on the f_j+1-st execution is $q_j(f_j) = q_j^{f_j}(1 - q_j)$. Assuming that errors occur independently for different jobs, the probability that a failure scenario $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ happens can then be computed as $Q(\mathbf{f}) = \prod_{j=1}^{n} q_j(f_j)$.

In general, given the probability $Q(\mathbf{f})$ of each failure scenario $\mathbf{f}$, we can define the *expected approximation ratio* of an algorithm ALG for a job set $\mathcal{J}$ as follows[4]:

$$\mathbb{E}\left[\frac{T_{\text{ALG}}(\mathcal{J})}{T_{\text{OPT}}(\mathcal{J})}\right] = \sum_{\mathbf{f}} Q(\mathbf{f}) \cdot \frac{T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s})}{T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)} , \quad (4)$$

and the algorithm is said to be an *r-approximation in expectation* if its expected approximation ratio is at most r for any job set $\mathcal{J}$, i.e.,

$$\sup_{\mathcal{J}} \mathbb{E}\left[\frac{T_{\text{ALG}}(\mathcal{J})}{T_{\text{OPT}}(\mathcal{J})}\right] = r . \quad (5)$$

While the approximation ratio of a scheduling algorithm under any failure scenario shows its worst-case performance, the expected approximation ratio shows its average-case performance. Note that a worst-case ratio directly translates to the average case, because if the ratio holds for every failure scenario, it also holds for the weighted sum. However, the converse may not be the case: an algorithm could have a very good expected approximation ratio, but perform arbitrarily worse than the optimal in some (low probability) failure scenarios.

In the theoretical analysis (Section 4), we mainly focus on bounding the worst-case approximation ratios of the proposed algorithms (except in Section 4.6, where we study the average-case performance of the BATCH-LIST algorithm). For the experimental evaluations (Section 5), we will instantiate the failure model with the silent error probability for each job as defined in Equation (3), and report both worst-case and average-case performance of the algorithms under a variety of experimental scenarios.

4. While we use *expectation of ratios* to define the average-case performance of an algorithm, some studies in stochastic scheduling and online algorithms (e.g., [25], [29]) have used *ratio of expectations*, i.e.,

$$\frac{\mathbb{E}(T_{\text{ALG}})}{\mathbb{E}(T_{\text{OPT}})} = \frac{\sum_{\mathbf{f}} Q(\mathbf{f}) \cdot T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s})}{\sum_{\mathbf{f}} Q(\mathbf{f}) \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)} .$$

This approach, however, has not been favored by recent studies, since $\mathbb{E}(T_{\text{ALG}})$ could be dominated by "a few" instances with large objective functions, thus the ratio may not reflect the actual performance of the algorithm for "most" instances. See [33], [34] for a discussion on the two approaches.

4 Resilient Scheduling Algorithms

In this section, we present two resilient scheduling algorithms (LPA-LIST and BATCH-LIST), and derive their approximation ratios for some common speedup models. Note that, due to lack of space, some of the proofs can be found in the Web Supplementary Material (WSM).

4.1 A Lower Bound on the Makespan

We first consider a simple lower bound on the makespan of any scheduling algorithm under a given failure scenario. This generalizes the well-known lower bound [27], [36] for the failure-free case.

Let $\mathbf{p}$ denote the processor allocation decision made by a scheduling algorithm ALG for job set $\mathcal{J}$ under failure scenario $\mathbf{f}$. Then, we define, respectively, the *maximum cumulative execution time* and *total cumulative area* of the jobs under algorithm ALG to be:

$$t_{\max}(\mathcal{J},\mathbf{f},\mathbf{p}) = \max_{1\le j\le n}\sum_{i=1}^{f_j+1} t_j(p_j^{(i)}) \ , \tag{6}$$

$$A(\mathcal{J},\mathbf{f},\mathbf{p}) = \sum_{j=1}^{n}\sum_{i=1}^{f_j+1} a_j(p_j^{(i)}) \ . \tag{7}$$

The following quantity serves as a lower bound on the makespan of the algorithm for job set $\mathcal{J}$ under failure scenario $\mathbf{f}$:

$$L(\mathcal{J},\mathbf{f},\mathbf{p}) = \max\left(t_{\max}(\mathcal{J},\mathbf{f},\mathbf{p}), \frac{A(\mathcal{J},\mathbf{f},\mathbf{p})}{P}\right) . \tag{8}$$

Thus, we have:

$$T_{\text{ALG}}(\mathcal{J},\mathbf{f},\mathbf{p},\mathbf{s}) \ge L(\mathcal{J},\mathbf{f},\mathbf{p}) \ , \tag{9}$$

regardless of the scheduling decision $\mathbf{s}$ of the algorithm.

4.2 LPA-LIST Scheduling Algorithm

Our first algorithm, called LPA-LIST, adopts a *two-phase* approach [27], [36]. The first phase uses a *Local Processor Allocation* (LPA) strategy to decide processor allocation $\mathbf{p}$ of the jobs, and the second phase uses LIST scheduling to determine the starting time $\mathbf{s}$ of the jobs.

4.2.1 LIST *Scheduling Strategy*

We first discuss LIST scheduling for the second phase, assuming a given processor allocation $\mathbf{p}$. Algorithm 1 shows the pseudocode. The strategy first organizes all jobs in a list based on some priority. Then, at time 0 or whenever a running job J_k completes and hence releases processors, the algorithm detects if job J_k has errors. If so, the job will be inserted back into the list, again based on its priority, to be re-scheduled later. It finally scans the list of pending jobs and schedules all jobs that can be executed at the current time with the available processors. We point out that the algorithm essentially resembles a greedy backfilling strategy. In our analysis below, we will show that the worst-case approximation ratio is independent of the job priorities used, although it may affect the algorithm's practical performance. In Section 5, we will consider some commonly used priority rules for the experimental evaluation.

Algorithm 1: LIST (Scheduling Strategy)

```
Organize all jobs in a list $L$ according to some priority rule;
$P_{avail} \leftarrow P$;
$f_j \leftarrow 0, \forall j$;
when at time 0 or a running job $J_k$ completes execution do
    $P_{avail} \leftarrow P_{avail} + p_k^{(f_k+1)}$;
    if job $J_k$ failed then
        $L.insert\_with\_priority(J_k)$;
        $f_k \leftarrow f_k + 1$;
    end
    for $j = 1, \dots, |L|$ do
        $J_j \leftarrow L(j)$;
        if $P_{avail} \ge p_j^{(f_j+1)}$ then
            execute job $J_j$ at the current time;
            $P_{avail} \leftarrow P_{avail} - p_j^{(f_j+1)}$;
            $L.remove(J_j)$;
        end
    end
end
```

The following lemma shows the worst-case performance of the LIST scheduling strategy. Note that the job set $\mathcal{J}$ is dropped from the notations since the context is clear.

Lemma 1. *Given a processor allocation decision* $\mathbf{p}$ *for the jobs, the makespan of a* LIST *schedule (that determines the starting times* $\mathbf{s}$*) under any failure scenario* $\mathbf{f}$ *satisfies:*

$$T_{\text{LIST}}(\mathbf{f},\mathbf{p},\mathbf{s}) \le \begin{cases} \frac{2A(\mathbf{f},\mathbf{p})}{P}, & \text{if } p_{\min} \ge \frac{P}{2} \\ \frac{A(\mathbf{f},\mathbf{p})}{P-p_{\min}} + \frac{(P-2p_{\min})\cdot t_{\max}(\mathbf{f},\mathbf{p})}{P-p_{\min}}, & \text{if } p_{\min} < \frac{P}{2} \end{cases}$$

where $p_{\min} \ge 1$ *denotes the minimum number of utilized processors at any time during the schedule.*

Proof. This proof builds on the observation that LIST only allocates and de-allocates processors upon job completions. Hence, the entire schedule can be divided into a set of consecutive and non-overlapping intervals $\mathcal{I} = \{I_1, I_2, \dots, I_v\}$, where jobs start (or complete) at the beginning (or end) of an interval, and v denotes the total number of intervals. Then, a case study on the value of $p_{\min}$ leads to the result (see the Web Supplementary Material (WSM)). □

While Lemma 1 bounds the general performance of a LIST schedule for a given processor allocation $\mathbf{p}$, the following lemma shows its approximation ratio when the processor allocation strategy satisfies certain properties.

Lemma 2. *Given any failure scenario* $\mathbf{f}$*, if the processor allocation decision* $\mathbf{p}$ *satisfies:*

$$A(\mathbf{f},\mathbf{p}) \le \alpha \cdot A(\mathbf{f},\mathbf{p}^*) \ ,$$
$$t_{\max}(\mathbf{f},\mathbf{p}) \le \beta \cdot t_{\max}(\mathbf{f},\mathbf{p}^*) \ ,$$

where $\mathbf{p}^*$ *denotes the processor allocation of an optimal schedule, then a* LIST *schedule using processor allocation* $\mathbf{p}$ *is* $r(\alpha,\beta)$*-approximation, where*

$$r(\alpha,\beta) = \begin{cases} 2\alpha, & \text{if } \alpha \ge \beta \\ \frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta, & \text{if } \alpha < \beta \end{cases} \tag{10}$$

Proof. This proof builds on Lemma 1, see the WSM. □

4.2.2 *Local Processor Allocation (*LPA*)*

We now discuss the LPA strategy for the first phase of the algorithm. Given the result of Lemma 2, LPA allocates processors locally for each job. Algorithm 2 shows its pseudocode. For each job J_j, the strategy first computes its

Algorithm 2: LPA (Processor Allocation Strategy)

```
for $j = 1, 2, \ldots, n$ do
    $t_{\min} \leftarrow \infty, a_{\min} \leftarrow \infty$;
    for $p = 1, 2, \ldots, P$ do
        if $t_j(p) < t_{\min}$ then
            $t_{\min} \leftarrow t_j(p)$;
        end
        if $p \cdot t_j(p) < a_{\min}$ then
            $a_{\min} \leftarrow p \cdot t_j(p)$;
        end
    end
    $p_j \leftarrow 0, r_{\min} \leftarrow \infty$;
    for $p = 1, 2, \ldots, P$ do
        $\alpha \leftarrow p \cdot t_j(p)/a_{\min}$;
        $\beta \leftarrow t_j(p)/t_{\min}$;
        compute $r(\alpha, \beta)$ from Equation (10);
        if $r(\alpha, \beta) < r_{\min}$ then
            $p_j \leftarrow p$, $r_{\min} \leftarrow r(\alpha, \beta)$;
        end
    end
end
```

minimum possible execution time and area. Then, it chooses a processor allocation that leads to the smallest ratio $r(\alpha, \beta)$ defined in Equation (10) based on the job's local bounds (α and β) on the area and execution time. If all jobs satisfy the same bounds, then the bound will also hold globally.

Once the processor allocation of a job has been decided, the same allocation will be used by the LIST scheduling strategy in the second phase throughout the execution until the job completes successfully without failures.

4.3 Worst-Case Performance of LPA-LIST for Some Common Speedup Models

We now analyze the worst-case performance of the LPA-LIST algorithm for moldable jobs that exhibit some common speedup models, as well as for the general monotonic model. All derived approximation ratios are independent of the failure scenarios, hence based on Equations (4) and (5). The same ratios also apply to the average-case performance of the algorithm for the respective speedup models.

4.3.1 Roofline Model

In the roofline model, the execution time of a job J_j when allocated p processors satisfies $t_j(p) = \frac{w_j}{\min(p, \bar{p}_j)}$ for a bounded degree of parallelism $1 \leq \bar{p}_j \leq P$.

Theorem 1. LPA-LIST *is a 2-approximation for jobs with the roofline model, and this bound is tight.*

Proof. In the roofline speedup model, the minimum execution time of a job J_j is $t_{\min} = w_j/\bar{p}_j$ and the minimum area of the job is $a_{\min} = w_j$. These two quantities can be achieved by simply allocating $p_j = \bar{p}_j$ processors to the job. This leads to the bounds of $\alpha = 1$ and $\beta = 1$ for each job as well as globally under any failure scenario. Hence, based on Lemma 2, we get an approximation ratio of $2\alpha = 2$.

To show that this bound is tight, suppose $P = 2K$ for some $K > 0$, and consider two identical jobs with $w = K+1$ and $\bar{p} = K + 1$. Allocating $K + 1$ processors achieves $\alpha = \beta = 1$, thus clearly minimizing the ratio $r(\alpha, \beta)$ for both jobs. Suppose jobs do not fail. The makespan will then be $T = 2$, since the two jobs must be processed sequentially one after the other. However, the optimal algorithm would allocate K processors to both jobs and execute them in parallel, resulting in a makespan of $T_{\text{OPT}} = 1 + \frac{1}{K}$. This gives an approximation ratio of $\frac{T}{T_{\text{OPT}}} = \frac{2}{1+\frac{1}{K}} = 2 - \frac{2}{K+1}$, which can be arbitrarily close to 2 when K is large enough. $\square$

4.3.2 Communication Model

In the communication model [11], [16], the execution time of a job J_j when allocated p processors is given by $t_j(p) = w_j/p + (p-1)c_j$, where $c_j \geq 0$ denotes the per-processor communication overhead.

Theorem 2. LPA-LIST *is a 3-approximation for jobs with the communication model.*

Proof. For the communication model, we consider a processor allocation p_j for a job J_j and show that it achieves $\alpha = \beta = \frac{3}{2}$, i.e., $a_j(p_j) \leq \frac{3}{2}a_{\min}$ and $t_j(p_j) \leq \frac{3}{2}t_{\min}$. Hence, based on Lemma 2, we get an approximation ratio of $2\alpha = 3$. The detailed proof can be found in the WSM. $\square$

Remarks. Our result improves upon the 4-approximation of the SET algorithm [16], which is the best ratio known for the communication model[5]. Our result further extends the one in [16] in two ways: (1) The model in [16] assumes the same communication overhead c for all jobs, while we consider an individual overhead c_j for each job J_j; (2) The algorithm in [16] applies to failure-free job executions, while our algorithm is able to handle job failures.

Theorem 3. *The approximation ratio of* LPA-LIST *is at least 2.5 for jobs with the communication model.*

The proof can be found in the WSM.

4.3.3 Amdahl's Model

In Amdahl's model [1], the execution time of a job J_j when allocated p processors satisfies $t_j(p) = w_j(\frac{1-\gamma_j}{p} + \gamma_j)$, where $\gamma_j \in [0, 1]$ denotes the inherently sequential fraction of the job. It is a particular case of the monotonic model as described in Section 3.1. For convenience, we consider an equivalent form of the model in the analysis: $t_j(p) = \frac{w_j}{p} + d_j$, where w_j denotes the parallelizable work of the job and d_j denotes the inherently sequential work.

Theorem 4. LPA-LIST *is a 4-approximation for jobs with the Amdahl's model.*

Proof. In Amdahl's model, the minimum execution time of a job J_j is $t_{\min} = \frac{w_j}{P} + d_j$ (achieved by allocating P processors), and the minimum area of the job is $a_{\min} = w_j + d_j$ (achieved by allocating one processor). We consider a processor allocation of $p_j = \min(\lceil \frac{w_j}{d_j} \rceil, P)$ for the job.

For the area, we have $a_j(p_j) = w_j + p_j d_j \leq w_j + \lceil \frac{w_j}{d_j} \rceil d_j \leq w_j + (\frac{w_j}{d_j} + 1)d_j = 2w_j + d_j \leq 2a_{\min}$. Thus, we get $\alpha = 2$.

For the execution time, we consider two cases: (1) If $\lceil \frac{w_j}{d_j} \rceil \leq P$, then $p_j = \lceil \frac{w_j}{d_j} \rceil$, and we have $t_j(p_j) = \frac{w_j}{p_j} + d_j \leq \frac{w_j}{w_j/d_j} + d_j = 2d_j \leq 2t_{\min}$. In this case, we get $\beta = 2$; (2) If $\lceil \frac{w_j}{d_j} \rceil > P$, then $p_j = P$, and we have $t_j(p_j) = \frac{w_j}{P} + d_j = t_{\min}$. In this case, we get $\beta = 1$.

Hence, based on Lemma 2, we get an approximation ratio of $2\alpha = 4$. $\square$

5. The SET algorithm [16] minimizes the execution time of each job, resulting in $\alpha = 2$ and $\beta = 1$, hence an approximation ratio of $2\alpha = 4$.

Theorem 5. *The approximation ratio of* LPA-LIST *is at least* 3 *for jobs with the Amdahl's model.*

The proof can be found in the WSM.

4.3.4 Mix Model

We now consider the mix model combining Roofline, Communication and Amdahl's models as follows: $t_j(p) = \frac{w_j(1-\gamma_j)}{\min(p,\bar{p}_j)} + w_j\gamma_j + (p-1)c_j$, which could capture more realistically the speedups of some complex applications. In this model, we only need to consider $p \le \bar{p}_j$, since any $p > \bar{p}_j$ will obviously be a bad choice. To simplify the analysis, and since we assume that all parameters are strictly positive, we can factorize the function by c_j and obtain the following equivalent form: $t_j(p) = c_j\left(\frac{w'_j}{p} + d'_j + (p-1)\right)$, with $w'_j = \frac{w_j(1-\gamma_j)}{c_j}$ and $d'_j = \frac{w_j\gamma_j}{c_j}$.

Theorem 6. LPA-LIST *is a 6-approximation for jobs with the mix model.*

Proof. For this mix model, we provide a processor allocation p_j for a job J_j and show that it achieves $\alpha = \beta = 3$, i.e., $a_j(p_j) \le 3a_{\min}$ and $t_j(p_j) \le 3t_{\min}$. Hence, based on Lemma 2, we get an approximation ratio of $2\alpha = 6$. The detailed proof can be found in the WSM. $\square$

Theorem 7. *The approximation ratio of* LPA-LIST *is at least* 3 *for jobs with the mix model.*

Proof. When c_j is small enough, the mix model can get arbitrarily close to Amdahl's model, for which LPA-LIST is at least a 3-approximation. $\square$

4.3.5 Power Model

In the power model, the execution time of a job J_j when allocated p processors satisfies $t_j(p) = w_j/p^{\delta_j}$, where $\delta_j \in [0,1]$ is a constant parameter. This speedup has been observed in some linear algebra applications [14], [35] and it is also an example of the monotonic model.

Theorem 8. LPA-LIST *is a* $\Theta(P^{1/4})$*-approximation for jobs with the power model.*

Proof. In the power model, the minimum execution time of a job J_j is $t_{\min} = \frac{w_j}{P^{\delta_j}}$ (achieved by allocating P processors), and the minimum area of the job is $a_{\min} = w_j$ (achieved by allocating one processor).

We consider a processor allocation of $p_j = \lceil P^{\delta_j}\rceil$. In this case, we get $\alpha = \frac{a_j(p_j)}{a_{\min}} = p_j^{1-\delta_j} \ge P^{\delta_j(1-\delta_j)}$ and $\beta = \frac{t_j(p_j)}{t_{\min}} = (\frac{P}{p_j})^{\delta_j} \le P^{\delta_j(1-\delta_j)} \le \alpha$. Thus, based on Lemma 2, we get an approximation ratio of $2\alpha = 2\lceil P^{\delta_j}\rceil^{1-\delta_j} < 2(P^{\delta_j}+1)^{1-\delta_j} < 2(P^{\delta_j(1-\delta_j)}+1)$. The last inequality is because $(x+1)^\mu < x^\mu + 1$ for any $x > 0$ and $0 < \mu < 1$. Furthermore, the value of $\delta_j(1-\delta_j)$ is maximized at $\delta_j = 1/2$. This results in an approximation ratio of $2(P^{1/4}+1)$.

To show that the above ratio is asymptotically tight for the algorithm, consider a single job with $\delta_j = 1/2$, so we have $\alpha = p_j^{1/2}$ and $\beta = (\frac{P}{p_j})^{1/2}$. Clearly, LPA-LIST will allocate at most $P^{1/2}$ processors to the job; otherwise, we would have $\alpha > \beta$, and according to Lemma 2, the ratio $r(\alpha,\beta) = 2\alpha = 2p_j^{1/2}$ will increase with the processor allocation. Thus, the execution time of the job under LPA-LIST will be at least $T \ge \frac{w_j}{P^{1/4}}$, whereas the optimal execution time is $T_{\text{OPT}} = \frac{w_j}{P^{1/2}}$ by allocating P processors. This gives an approximation ratio $\frac{T}{T_{\text{OPT}}} \ge P^{1/4}$. $\square$

4.3.6 Monotonic Model

We now consider the general monotonic model. Recall that a job J_j is *monotonic*, if $t_j(p) \ge t_j(p')$ and $a_j(p) \le a_j(p')$ for any $p \le p'$. This means that the execution time of the job will not increase with the processor allocation and the area will not decrease with the processor allocation. In particular, the area assumption implies that the speedup efficiency of the job will not increase as more processors are allocated to it, i.e., $\sigma_j(p)/p \ge \sigma_j(p')/p'$, a property that has been observed in many practical parallel applications.

Theorem 9. LPA-LIST *is an* $O(\sqrt{P})$*-approximation for jobs with the monotonic model.*

Proof. In a general monotonic model, the minimum execution time of a job J_j is achieved with P processors, i.e., $t_{\min} = t_j(P)$, and the minimum area is achieved with one processor, i.e., $a_{\min} = a_j(1) = t_j(1)$.

Consider an allocation $p_j = \lfloor\sqrt{P}\rfloor$. Based on the monotonic assumption, we get $a_j(p_j) = p_j t_j(p_j) \le \sqrt{P}\cdot t_j(1) = \sqrt{P}\cdot a_{\min}$, and $t_j(p_j) \le \frac{P}{p_j}t_j(P) = O(\sqrt{P})\cdot t_{\min}$. Thus, based on Lemma 2, we get an approximation ratio of $O(\sqrt{P})$. $\square$

We show that the above ratio is asymptotically tight for any algorithm that makes *local* processor allocation decisions based on individual job characteristics. Examples of such algorithms include the LPA algorithm considered in this paper and the SET algorithm studied in [16]. The result holds even under the additional assumption that the speedup profiles of the jobs are *concave* [22] and that jobs do not fail. In the next section, we will propose another algorithm that overcomes this limitation by making *coordinated* processor allocation decisions for a set of jobs.

Theorem 10. *Any scheduling algorithm that relies on local processor allocation for each individual job is* $\Omega(\sqrt{P})$*-approximation for jobs with the monotonic model.*

Proof. Assume that $\sqrt{P}$ is an integer and $P \ge 4$. We consider a job with a concave speedup profile[6] that contains two piece-wise linear segments defined by three points: $\sigma(1) = 1$, $\sigma(\sqrt{P}) = 2$ and $\sigma(P) = \sqrt{P}$ (see Figure 1(a)). Suppose the execution time of the job with one processor is $t(1) = 1$. We can then derive the execution time profile of the job as follows (see Figure 1(b)):

$$t(p) = \begin{cases} \frac{\sqrt{P}-1}{p+\sqrt{P}-2} & \text{if } p \le \sqrt{P}, \\ \frac{P-\sqrt{P}}{p(\sqrt{P}-2)+P} & \text{if } p > \sqrt{P}; \end{cases}$$

and the area profile of the job as follows (see Figure 1(c)):

$$a(p) = \begin{cases} \frac{p(\sqrt{P}-1)}{p+\sqrt{P}-2} & \text{if } p \le \sqrt{P}, \\ \frac{p(P-\sqrt{P})}{p(\sqrt{P}-2)+P} & \text{if } p > \sqrt{P}. \end{cases}$$

6. The speedup profile is concave because $\sigma'(p) = \frac{1}{\sqrt{P}-1}$ for any $p \in [1,\sqrt{P})$, and $\sigma'(p) = \frac{\sqrt{P}-2}{P-\sqrt{P}} < \frac{\sqrt{P}}{P-\sqrt{P}} = \frac{1}{\sqrt{P}-1}$ for any $p \in (\sqrt{P}, P]$.

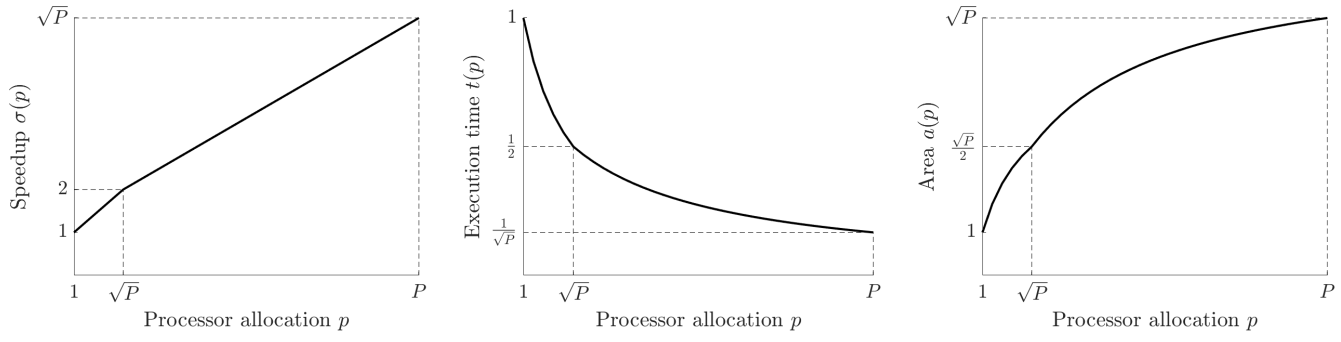


Fig. 1. (a) Speedup, (b) execution time, and (c) area profiles of the job used in the proof of Theorem 10.

The job is obviously monotonic.

Suppose there are n identical such jobs in the system, where n depends on the processor allocation algorithm (denoted as ALG). Since the jobs are identical and processors are allocated locally, the processor allocation p for each job should be the same. We consider two cases.

Case 1: If $p \leq \sqrt{P}$, then there is only $n = 1$ job. In this case, the algorithm has a makespan of $T_{\text{ALG}} \geq t(\sqrt{P}) = \frac{1}{2}$ and the optimal makespan is $T_{\text{OPT}} = t(P) = \frac{1}{\sqrt{P}}$ by allocating P processors to the job.

Case 2: If $p > \sqrt{P}$, then there are $n = P$ jobs. In this case, the makespan of the algorithm satisfies $T_{\text{ALG}} \geq \frac{n \cdot a(p)}{P} \geq a(\sqrt{P}) = \frac{\sqrt{P}}{2}$, and the optimal makespan is $T_{\text{OPT}} = 1$ by allocating one processor to each job.

Thus, in both cases, we have $\frac{T_{\text{ALG}}}{T_{\text{OPT}}} \geq \frac{\sqrt{P}}{2}$. $\square$

4.4 BATCH-LIST Scheduling Algorithm

We now present the second algorithm, called BATCH-LIST. Unlike the LPA-LIST algorithm, which allocates processors locally for each job, BATCH-LIST coordinates the processor allocation decisions for different jobs. While not knowing the failure scenario in advance, the algorithm organizes the execution attempts of the jobs in multiple *batches*, where each batch executes the pending jobs (i.e., the jobs that have not been successfully completed so far) up to a certain number of attempts that doubles after each batch. The idea is inspired by the *doubling strategy* [9] that has been commonly applied in many online problems. The following describes the details of the BATCH-LIST algorithm.

Let B_k denote the k-th batch created by the algorithm, where $k \geq 1$. Let n_k denote the number of pending jobs immediately before B_k starts, and let $\mathcal{J}_k = \{J_{k,1}, J_{k,2}, \dots, J_{k,n_k}\}$ denote this set of pending jobs. For convenience, we define $g_k = 2^{k-1}$. In batch B_k, we allow each pending job $J_{k,j}$ to have at most $f_{k,j} = g_k - 1$ failures, i.e., each job is allowed to make g_k execution attempts in the batch; if the job is still not successfully completed after that, it will be handled by the next batch B_{k+1}. Let $\mathbf{f}_k = (f_{k,1}, f_{k,2}, \dots, f_{k,n_k})$ denote this worst-case failure scenario for the jobs in batch B_k. Given $\mathbf{f}_k$, each job $J_{k,j}$ can be represented by a chain $J_{k,j}^{(1)} \rightarrow J_{k,j}^{(2)} \rightarrow \cdots \rightarrow J_{k,j}^{(g_k)}$ of g_k sub-jobs with linear precedence constraint, where each sub-job represents an execution attempt of $J_{k,j}$ in the batch. Thus, all sub-jobs in batch B_k form a set of n_k linear chains, one for each pending job.

To allocate processors for all the sub-jobs (or the different execution attempts of the pending jobs) in batch B_k, we adopt the pseudo-polynomial time algorithm, called MT-ALLOTMENT, proposed in [26] for series-parallel precedence graphs (of which a set of independent linear chains is a special case). Specifically, the algorithm determines an allocation $p_{k,j}^{(m)}$ for each sub-job $J_{k,j}^{(m)}$ (or the m-th execution attempt of job $J_{k,j}$). Let $\vec{p}_{k,j} = (p_{k,j}^{(1)}, p_{k,j}^{(2)}, \dots, p_{k,j}^{(f_{k,j}+1)})$ be the vector of processor allocations for job $J_{k,j}$, and let $\mathbf{p}_k = (\vec{p}_{k,1}, \vec{p}_{k,2}, \dots, \vec{p}_{k,n_k})$ be the processor allocations for all jobs in batch B_k. The following lemma shows the property of the allocation $\mathbf{p}_k$ returned by MT-ALLOTMENT for jobs with any arbitrary speedup model.

Lemma 3. *For any $\epsilon > 0$, MT-ALLOTMENT can compute, with complexity polynomial in $1/\epsilon$, a processor allocation $\mathbf{p}_k$ for all jobs in batch B_k that approximates the minimum makespan lower bound as defined in Equation (8) as follows:*

$$L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}_k) \leq (1+\epsilon) \cdot \min_{\mathbf{p}} L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}) . \tag{11}$$

We refer to [26] for a detailed description of the MT-ALLOTMENT algorithm and its analysis[7]. Once the processor allocation $\mathbf{p}_k$ has been decided, BATCH-LIST schedules all pending jobs in a batch B_k using the LIST strategy as shown in Algorithm 1, while restricting each job to execute at most g_k times. After batch B_k completes and if there are still pending jobs, the algorithm will create a new batch B_{k+1} to schedule the remaining pending jobs.

4.5 Worst-Case Performance of BATCH-LIST for Arbitrary Speedup Model

We analyze the worst-case performance of BATCH-LIST for moldable jobs with any arbitrary speedup model.

First, we define the following concept: a job set $\mathcal{J}'$ with failure scenario $\mathbf{f}'$ is said to be *dominated* by a job set $\mathcal{J}$ with failure scenario $\mathbf{f}$, denoted by $(\mathcal{J}', \mathbf{f}') \subseteq (\mathcal{J}, \mathbf{f})$, if for every job $J_j \in \mathcal{J}'$, we have $J_j \in \mathcal{J}$ and $f'_j \leq f_j$. The following lemma gives two trivial properties without proof for a dominated pair of job set and failure scenario.

Lemma 4. *If $(\mathcal{J}', \mathbf{f}') \subseteq (\mathcal{J}, \mathbf{f})$, then we have:*
(a) $L(\mathcal{J}', \mathbf{f}', \mathbf{p}) \leq L(\mathcal{J}, \mathbf{f}, \mathbf{p})$;

7. In a nutshell, the algorithm uses dynamic programming to decide whether there exists an allocation $\mathbf{p}$ such that $L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}) \leq (1+\epsilon) \cdot X$ for a positive integer bound X, and performs a binary search on X.

(b) $T_{\text{OPT}}(\mathcal{J}', \mathbf{f}', \mathbf{p}'^*, \mathbf{s}'^*) \leq T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*)$.

Lemma 5. *Suppose a job set $\mathcal{J}$ with failure scenario $\mathbf{f}$ is executed by* BATCH-LIST. *Then, any job $J_j \in \mathcal{J}$ will successfully complete in $b_j = \lceil \log_2(f_j + 2) \rceil$ batches, and in any batch B_k, where $1 \leq k \leq b_j$, we have $f_{k,j} \leq f_j$.*

Proof. Since the algorithm allows the number of execution attempts of a job to double in each new batch, the maximum number of execution attempts of the job in a total of b batches is given by $\sum_{k=1}^{b} 2^{k-1} = 2^b - 1$. Thus, if a job J_j fails f_j times (i.e., executes $f_j + 1$ times), then the number of batches it takes to complete the job is $b_j = \lceil \log_2(f_j + 2) \rceil = 1 + \lfloor \log_2(f_j + 1) \rfloor$.

In any batch B_k until job J_j completes, where $1 \leq k \leq b_j$, we have $f_{k,j} = 2^{k-1} - 1 \leq 2^{\lfloor \log_2(f_j+1) \rfloor} - 1 \leq f_j$. □

The following theorem shows the approximation ratio of BATCH-LIST for jobs with arbitrary speedup model.

Theorem 11. BATCH-LIST *is an $O((1 + \epsilon) \log_2(f_{\max}))$-approximation for jobs with arbitrary speedup model, where $f_{\max} = \max_j f_j$ denotes the maximum number of failures of any job in a failure scenario.*

Proof. According to Lemma 5, the total number of batches for any job set $\mathcal{J}$ with failure scenario $\mathbf{f}$ is given by $b_{\max} = \lceil \log_2(f_{\max} + 2) \rceil$. Further, for any batch B_k, where $1 \leq k \leq b_{\max}$, we have $(\mathcal{J}_k, \mathbf{f}_k) \subseteq (\mathcal{J}, \mathbf{f})$.

Let $\mathbf{f}'_k = (f'_{k,1}, f'_{k,2}, \dots, f'_{k,n_k})$ denote the actual failure scenario for the jobs in batch B_k. Clearly, we have $f'_{k,j} \leq f_{k,j}$ for any $J_j \in \mathcal{J}_k$, and thus, $(\mathcal{J}_k, \mathbf{f}'_k) \subseteq (\mathcal{J}_k, \mathbf{f}_k)$.

Since BATCH-LIST uses the MT-ALLOTMENT algorithm to allocate processors and the LIST strategy to schedule all jobs in each batch, according to Lemmas 1, 3 and 4, we can bound the execution time of any batch B_k as follows:

$$\begin{aligned}
T_{\text{LIST}}(\mathcal{J}_k, \mathbf{f}'_k, \mathbf{p}_k, \mathbf{s}_k) &\leq 2 \cdot L(\mathcal{J}_k, \mathbf{f}'_k, \mathbf{p}_k) \\
&\leq 2 \cdot L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}_k) \\
&\leq 2(1+\epsilon) \cdot L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}^*_k) \\
&\leq 2(1+\epsilon) \cdot T_{\text{OPT}}(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}^*_k, \mathbf{s}^*_k) \\
&\leq 2(1+\epsilon) \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*) \ .
\end{aligned}$$

Therefore, the makespan of BATCH-LIST satisfies:

$$\begin{aligned}
T_{\text{BATCH-LIST}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) &= \sum_{k=1}^{b_{\max}} T_{\text{LIST}}(\mathcal{J}_k, \mathbf{f}'_k, \mathbf{p}_k, \mathbf{s}_k) \\
&\leq 2(1+\epsilon) \lceil \log_2(f_{\max} + 2) \rceil \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*) \ . \quad \square
\end{aligned}$$

We now show that the approximation ratio of BATCH-LIST is tight up to a constant factor.

Theorem 12. BATCH-LIST *is $\Omega(\log_2(f_{\max}))$-approximation.*

Proof. We consider a set $\mathcal{J} = \{J_1, J_2, \dots, J_K\}$ of K jobs and at least as many processors, so that each job can be executed on a dedicated processor. For each job J_j, where $1 \leq j \leq K$, its (sequential) execution time is $t_j = \frac{1}{2^j}$, and it fails $f_j = 2^{j-1} - 1$ times (i.e., executes 2^{j-1} times). Given this failure scenario $\mathbf{f}$, the total time to complete job J_j is given by $2^{j-1} \cdot \frac{1}{2^j} = \frac{1}{2}$. The optimal makespan for this failure scenario is therefore $T_{\text{OPT}}(\mathcal{J}, \mathbf{f}) = \frac{1}{2}$.

In the above failure scenario, the maximum number of failures of any job is $f_{\max} = f_K = 2^{K-1} - 1$. Based on Lemma 5, BATCH-LIST will complete each job J_j in $\lceil \log_2(f_j + 2) \rceil = j$ batches, and will complete all jobs in $\lceil \log_2(f_{\max} + 2) \rceil = K$ batches. Figure 2 illustrates the execution of this failure scenario for $K = 5$. In each batch B_k, where $1 \leq k \leq K$, the set of pending jobs is given by $\mathcal{J}_k = \{J_k, J_{k+1}, \dots, J_K\}$. For the first batch B_1, it takes $t_1 = \frac{1}{2}$ time to complete job J_1 and thus the entire batch. For any batch B_k, where $2 \leq k \leq K - 1$, it takes $t_{k+1} = \frac{1}{2^{(k+1)}}$ time for each execution attempt of job J_{k+1}, which will have 2^{k-1} execution attempts. Thus, batch B_k will take $2^{k-1} \cdot \frac{1}{2^{(k+1)}} = \frac{1}{4}$ time to complete. The makespan of BATCH-LIST for the entire job set $\mathcal{J}$ then satisfies:

$$\begin{aligned}
T_{\text{BATCH-LIST}}(\mathcal{J}, \mathbf{f}) &\geq \frac{1}{2} + (K-2) \cdot \frac{1}{4} \\
&= \frac{K}{4} = \frac{\lceil \log_2(f_{\max} + 2) \rceil}{2} \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f}) \ . \ \square
\end{aligned}$$

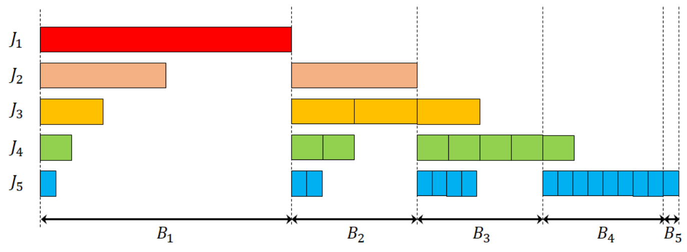


Fig. 2. An illustration of the lower bound instance for the BATCH-LIST algorithm shown in Theorem 12 with $K = 5$ jobs.

4.6 A Lower Bound on the Average-Case Performance of BATCH-LIST

The preceding section shows that the worst-case approximation ratio of BATCH-LIST grows linearly with the number b of batches. However, when jobs have fixed failure probabilities, the probability of having b batches tends to 0 as b approaches infinity. Thus, one might expect a constant approximation in expectation. In this section, we show that it is not true by providing an $\omega(1)$ lower bound. Despite this negative result, the experimental evaluation (in Section 5) shows that the average-case performance of the algorithm is very close to the optimal under many practical settings. Deriving an upper bound on the average-case approximation ratio of BATCH-LIST remains an open question.

Theorem 13. *The expected approximation ratio of* BATCH-LIST *is $\omega(1)$, if all jobs have constant failure probabilities.*

The proof of this theorem can be found in the WSM. We point out that the above lower bound applies when the jobs' failure probabilities are either arbitrarily defined or related to their sequential execution times as defined in Equation (3). In fact, Theorem 13 holds generally true as long as the failure probability q_j of each job J_j is upper-bounded by a constant ρ, i.e., $q_j \leq \rho < 1$ for all $j = 1, \dots, n$.

4.7 An Illustrative Example

In this section, we provide a simple example to illustrate the behavior of the LPA-LIST and BATCH-LIST algorithms. The problem instance consists of a set $\{J_1, J_2, J_3, J_4\}$ of $n = 4$ jobs to be scheduled on $P = 4$ processors, assuming J_3 and J_4 will fail exactly once while J_1 and J_2 will not fail.

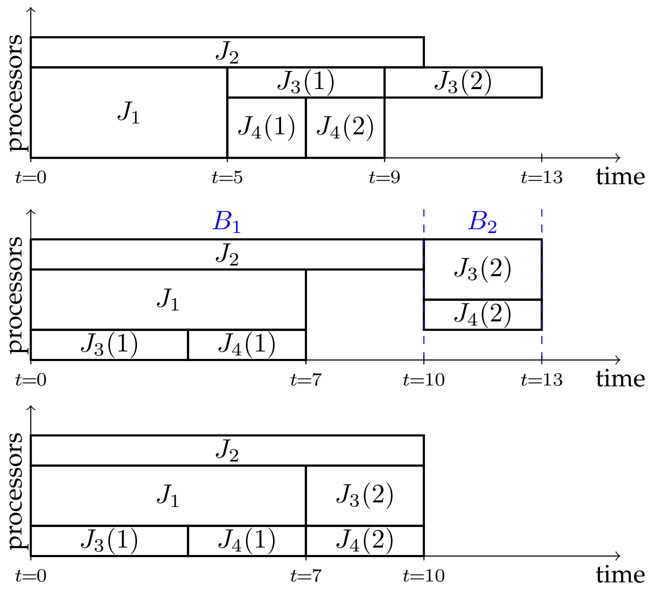


Fig. 3. An illustrative example showing the schedules produced by LPA-LIST (top), BATCH-LIST (middle), and the optimal algorithm (bottom).

The execution times of the jobs and the values of $r(\alpha,\beta)$ as functions of the processor allocation are given in the following table:

Job	$t(1)$	$t(2)$	$t(3)$	$t(4)$	$r(1)$	$r(2)$	$r(3)$	$r(4)$
J_1	11	7	5	4	3.2	2.9	2.7	2.9
J_2	10	9.8	9.6	9.5	2.04	3.9	5.8	7.6
J_3	4	3	3	2.5	2.4	3	4.5	5
J_4	3	2	1.7	1.4	2.76	2.73	3.4	3.7

Under these settings, LPA will allocate 3 processors to J_1 with an execution time of 5, 1 processor to J_2 with an execution time of 10, 1 processor to J_3 with an execution time of 4, and 2 processors to J_4 with an execution time of 2, as highlighted in red in the table. The jobs will then be scheduled using the LIST strategy. Assuming that jobs are prioritized from J_1 to J_4, the schedule produced by the LPA-LIST algorithm is shown in Figure 3 (top), with a makespan of 13. The main drawback of this algorithm is that the allocation of each job is fixed without considering the state of the system nor the remaining jobs to be scheduled at runtime. For instance, at time $t = 9$, only J_3 remains to be scheduled, so we could easily speed it up by allocating more processors: using the other two available processors would reduce the makespan by 1.

BATCH-LIST, on the contrary, will first schedule the four jobs in a batch B_1, assuming they will not fail. In this case, it will be able to find processor allocations that minimize the makespan lower bound by setting a small ϵ. Specifically, it will allocate 2 processors to J_1 and 1 processor to all the other jobs, resulting in an optimal lower bound of 10. The execution time for this first batch will also match this lower bound by using the LIST strategy to schedule the jobs. After the execution of the first batch, BATCH-LIST will plan for both failed jobs J_3 and J_4 to be executed two more times in a second batch B_2. The processor allocations in this batch that minimize the makespan lower bound would be 2 processors for J_3 and 1 processor for J_4 for both of their potential executions. However, the two jobs complete successfully after one execution attempt, giving an execution time of 3 for this second batch and an overall makespan of 13, as shown in Figure 3 (middle). In contrast to the LPA-LIST algorithm, the processor allocation of BATCH-LIST does take into account all available jobs in a batch, but the idle time between batches leads to an extra 3 units of time in makespan. Indeed, for this example, scheduling the second execution attempts of J_3 and J_4 as soon as possible would lead to an optimal schedule with a makespan of 10, as shown in Figure 3 (bottom).

5 PERFORMANCE EVALUATION

In this section, we evaluate and compare the performance of different scheduling algorithms using simulations on synthetic moldable jobs that follow various speedup models.

5.1 Simulation Setup

Evaluated Algorithms: We evaluate the performance of our two scheduling algorithms, namely, LPA-LIST (or LPA in short) and BATCH-LIST (or BATCH in short). For BATCH, we set $\epsilon = 0.3$ for its processor allocation procedure (Lemma 3). Their performance is also compared against that of the following two baseline heuristics:

- MINTIME: allocates processors to minimize the execution time of each job and schedules all jobs using the LIST strategy (Algorithm 1). This is also known as the shortest execution time (SET) algorithm in [16];
- MINAREA: allocates processors to minimize the area of each job and schedules all jobs using the LIST strategy.

Priority Rules: We consider three priority rules that have been shown to give good performance when (rigid) jobs are scheduled with the LIST strategy [5], which is used in all four evaluated algorithms (recall that BATCH uses LIST in each batch). The three priority rules are:

- LPT (Longest Processing Time): a job with a longer processing time has a higher priority;
- HPA (Highest Processor Allocation): a job with a higher processor allocation has a higher priority;
- LA (Largest Area): a job with a larger area has a higher priority.

Speedup Models: We generate synthetic moldable jobs that follow six speedup models: roofline, communication, Amdahl, mix (in two different versions) and power. Each job J_j is defined by two parameters: the total work w_j (i.e., the sequential execution time), which is drawn uniformly in $[5000, 4000000]$, and another parameter that depends on the speedup model.

- **Roofline**: the maximum degree of parallelism $\bar{p}_j$ is an integer drawn uniformly in $[100, 4000]$;
- **Communication**: the communication overhead is set as $c_j = \alpha \cdot 2^r$, where r is an integer uniformly chosen in $[0, 3]$ and α is drawn uniformly in $[1, 2]$.
- **Amdahl**: the sequential fraction is set as $\gamma_j = \frac{\alpha}{10^r}$, where r is an integer uniformly chosen in $[2, 7]$ and α is drawn uniformly in $[0, 10]$.
- **Mix**: we consider two different parameter settings: the first one, called **mix-low-com**, uses the same set of parameters as what is chosen for the roofline, communication, and Amdahl's model. The second one, called **mix**, uses $3c_j$ instead of c_j for the communication overhead.
- **Power**: the parameter δ_j is chosen uniformly in $[0, 1]$.

Failure Distribution: To generate failures for the jobs, we assume that silent errors follow the exponential distribution [17]. Let λ denote the error rate per unit of work, so a job will be struck by a silent error for every $1/\lambda$ unit of work executed on average. Following our failure model (Section 3), we assume parallelizing a job does not change the total number of computational operations (it may increase the communication, which we consider protected). Hence, the failure probability of a job will not depend on its processor allocation nor its execution time, but solely on its total work. For a job J_j with total work w_j, its failure probability is given by $q_j = 1 - e^{-\lambda w_j}$.

In the simulations, we set $\lambda = 10^{-7}$ by default. Given the chosen values of w_j, this corresponds to a failure probability between 0.0005 and 0.33 for a job. We also set the default number of processors and number of jobs to be $P = 7500$ and $n = 500$, but we will also vary all of these parameters to evaluate their impact on the performance.

Evaluation Methodology: The evaluation is done as follows: we generate 30 different sets of jobs, and for each set, 100 failure scenarios are drawn randomly from the failure distribution described above. For each of the failure scenarios, the simulated makespan of an algorithm is normalized by a lower bound (described below), which is then averaged over the 100 failure scenarios to estimate the expected ratio for the job set. Lastly, this ratio is averaged over the 30 job sets to compute the final expected performance of the algorithm. In addition, we also estimate the worst-case performance of the algorithm by using its largest normalized makespan over all job sets and failure scenarios.

Given job set $\mathcal{J}$ and a failure scenario $\mathbf{f}$, the makespan lower bound given in Equation (8) depends on the processor allocation and hence the scheduling algorithm. To ensure that the performance of all algorithms is normalized by the same quantity, we use the following rather loose lower bound, which is, however, independent of the scheduling decision:

$$L'(\mathcal{J}, \mathbf{f}) = \max\left(t'_{\max}(\mathcal{J}, \mathbf{f}), \frac{A'(\mathcal{J}, \mathbf{f})}{P}\right),$$

where $t'_{\max}(\mathcal{J}, \mathbf{f}) = \max_j \min_p (f_j + 1) t_j(p)$ is the minimum possible maximum execution time of all jobs, and $A'(\mathcal{J}, \mathbf{f}) = \sum_j \min_p (f_j + 1) a_j(p)$ is the minimum possible total area. Since this lower bound gives a pessimistic estimation on the optimal schedule, the actual performance of the algorithms is likely to be better than reported.

The simulation code for all experiments is publicly available at http://www.github.com/vlefevre/job-scheduling. Due to lack of space, we report here mainly results for the **mix** model, since it captures Roofline, Amdahl, and Communication as special cases. Full results can be found in the Web Supplementary Material (WSM).

5.2 Comparison of Algorithms and Priority Rules

We first compare the performance of different algorithms and study the impact of priority rules on their performance.

Figure 4 (top) shows the normalized makespans for the 11 combinations of algorithms and priority rules under the **mix** speedup model, with $\lambda = 10^{-7}$, $P = 7500$, and $n = 500$. For the MinArea algorithm, priority rules LA and LPT are identical, as the algorithm allocates one processor to all jobs, so only the results of LPT are reported. As we can see, MinArea fares poorly in most cases, because it allocates one processor to each job in order to minimize the area. This results in very long job execution (and re-execution) times, which leads to extremely large makespan. Moreover, allocating only one processor per job also results in idle processors thus resource inefficiency whenever the number of processors is higher than the number of jobs. The Lpa and Batch algorithms maintain a good balance between the execution time and area of a job, thus they perform well (and this remains true for all speedup models) in terms of both expected performance (bars) and worst-case performance (top endpoints of lines). Batch performs the best for the mix model. MinTime also performs relatively well with this set of parameters.

Figure 4 (bottom) further shows the results of four combinations of P and n with similar performance trends. We notice that these two parameters do have an impact on the performance of Batch, in particular at $P = 1000$ and $n = 500$. Indeed, when P is significantly larger than n, Batch tends to reduce all jobs to similar length and execute them at the same time, which gives the best tradeoff between the area and maximum execution time. In that case, the first batch, where all jobs are executed exactly once, is done almost perfectly. As the makespan of the first batch is dominant under $\lambda = 10^{-7}$, the overall makespan is closer to the lower bound. However, with $P = 1000$ and $n = 500$, there are not enough processors to execute all jobs at the same time. Thus, the performance of Batch becomes close to Lpa.

Note also that the performance of MinTime under the mix models becomes better when the number of processors is large compared to the number of jobs (e.g., $P = 10000, n = 100$). Indeed, MinTime is able to simultaneously minimize the execution time of all jobs in this case without using all the processors, thus achieving near-optimal performance. This is not possible with fewer processors, as minimizing the execution time alone for each job will increase the total area, which also plays an important role under such circumstance to have overall good performance.

Comparing the three priority rules, no significant difference is observed. In general, LPT and LA give similar results, and slightly better results than HPA. This is consistent with the results observed in [5] for scheduling rigid jobs. Given these results, we will only consider the LPT priority rule in the subsequent evaluation. We will also omit the MinArea algorithm, and focus on comparing the expected performance of the remaining algorithms.

5.3 Impact of Different Parameters

We now study the impact of different parameters on the performance of the algorithms. We start from $P = 7500$, $n = 500$, and $\lambda = 10^{-7}$, and vary one of these parameters in each experiment. We still focus on the mix model (recall that results for other speedup models are available in the WSM.

Impact of Number of Processors (P): Figure 5(a) shows the performance when the number of processors P is varied between 1000 and 15000. Batch outperforms Lpa despite the idle time at the end of each batch. This is due to Batch's

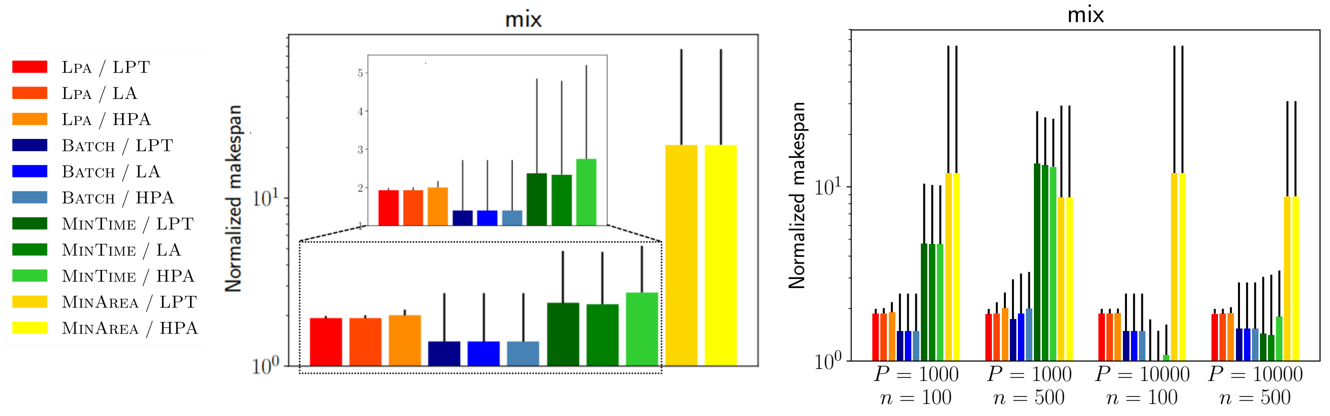


Fig. 4. Performance of different algorithms and priority rules under the **mix** model with $\lambda = 10^{-7}$, $P = 7500$, $n = 500$ (left) and four other different combinations of P and n (right). The bars represent expected performance and the top endpoints of the lines represent worst-case performance.

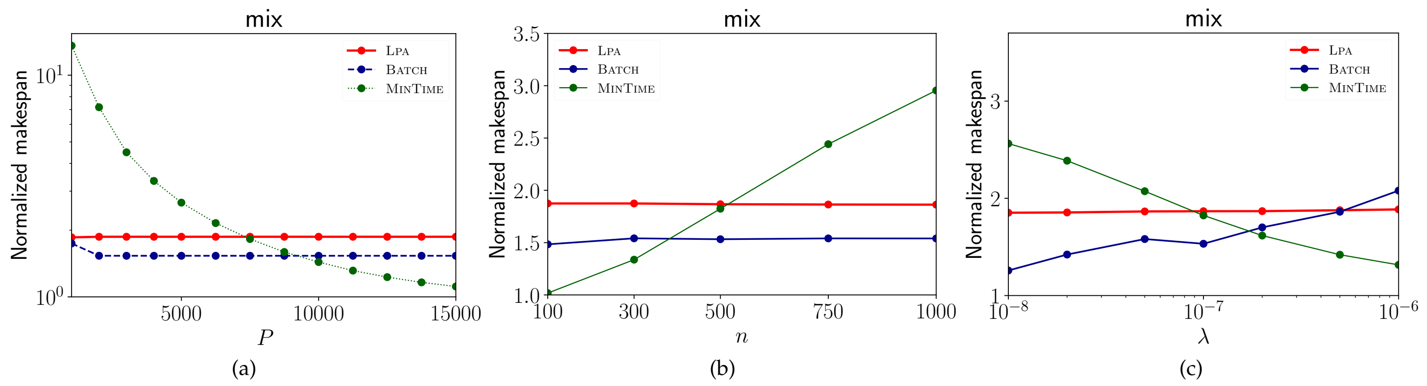


Fig. 5. Impact of (a) number of processors P, (b) number of jobs n, and (c) error rate λ on the performance of the algorithms for the **mix** model.

ability to better balance the job execution times globally, which becomes more important in this case. Moreover, the trend is not affected by the number of processors. Since both algorithms tend to allocate a relatively small number of processors for each job, the maximum degree of parallelism is not reached and the communication cost is relatively small. Note that the performance of MinTime is getting better with increasing number of processors. Indeed, the minimum execution time of a job is achieved with a reasonable number of processors because of the communication overhead. Thus, when P is high enough such that all jobs can be processed in parallel while minimizing their execution times, MinTime's allocation becomes close to optimal.

Impact of Number of Jobs (n): Figure 5(b) shows the performance when the number of jobs n is varied between 100 and 1000. Again, we can see that Batch has the best performance, except for small number of jobs. The number of jobs has a small impact for Batch, but only impacts MinTime as seen with varying P. Overall, as the number of jobs increases, the trend in the relative performance of the algorithms is consistent with the previous results we have observed in Figure 5(a) when the number of processors decreases.

Impact of Error Rate (λ): Figure 5(c) shows the impact of the error rate λ when it is varied between 10^{-8} (corresponding to 0.03 error per job on average) and 10^{-6} (corresponding to 12 errors per job on average). Once again, the relative performance of the three algorithms remains the same as before. While the performance of Lpa is barely affected, which is not surprising considering that its processor allocation is performed locally and separately from job scheduling, the performance of Batch gets worse with increasing error rate λ (and hence the number of failures), which corroborates the theoretical analysis (Theorem 11). In particular, when the error rate is small, there are very few failures and almost all jobs will complete in one batch. In this case, the processor allocation procedure of Batch (Lemma 3) is very precise. With increased error rate, more failures will occur and thus more batches will be introduced, causing scheduling inefficiencies from both idle times between the batches and possible imprecision in the processor allocations (especially with a large batch, since the actual number of failures may deviate significantly from the anticipated values). Finally, although the processor allocation is also performed locally for MinTime, the effect of increasing λ is similar to that of increasing P (or the opposite to that of increasing n): when there are more failures, we spend more time processing few large jobs that fail a lot, meaning that after some time only very few jobs are not finished yet. This effectively increases the total number of processors for these jobs or reduces the total number of jobs.

5.4 Summary of Results

Table 1 summarizes the makespan ratios of the four algorithms over the entire set of experiments, in terms of both average-case performance (expected ratio) and worst-case performance (maximum ratio). Overall, the results confirm the efficiency of our two resilient scheduling algorithms

Table 1. Summary of the performance for the four algorithms (with LPT priority rule) under the six speedup models.

Speedup Model		Roofline	Communication	Amdahl	Mix-low-com	Mix	Power
LPA	Expected	1.057	1.312	1.961	1.896	1.867	1.861
	Maximum	1.219	2.241	2.349	1.987	1.995	9.655
BATCH	Expected	1.158	1.434	1.529	1.548	1.571	1.549
	Maximum	1.999	2.449	2.874	3.674	4.164	3.975
MINTIME	Expected	1.057	2.044	15.567	2.810	2.704	20.386
	Maximum	1.219	2.666	49.795	12.611	27.174	61.726
MINAREA	Expected	114.079	122.199	23.594	16.875	9.686	2.571
	Maximum	1217.13	871.38	199.572	259.163	120.9	27.109

(LPA and BATCH), which outperform the baseline heuristics (MINTIME and MINAREA) in all settings. For the simplest roofline model, LPA is equivalent to MINTIME, both achieving a makespan very close to the lower bound (with a ratio around 1.06 on average). For the other models, we can observe significant performance difference between our best algorithm and the baseline. In particular, LPA achieves good performance with an expected ratio around 1.3 for the communication model, and an expected ratio less than 2 for the other models. We also notice that the maximum ratios are only slightly larger than the ones in the average case, and they remain much lower than those predicted by the theoretical bounds (except for the power model where the ratio is more than 9). BATCH also achieves excellent results thanks to its coordinated processor allocation and failure handling ability. It achieves a better average ratio (less than 1.6) for all models, but has larger worst-case ratios compared to LPA (except for the power model). On the other hand, the two baseline heuristics, although doing well in some scenarios, tend to have more irregular performance that depends on the model and parameter. In contrast, our algorithms exhibit more robust performance under various models and parameter settings.

6 Conclusion and Future Work

In this paper, we have studied the problem of scheduling moldable parallel jobs to cope with silent errors. We present a formal model of the problem and design two resilient scheduling algorithms (LPA and BATCH). While not knowing the failure scenarios of the jobs in advance, LPA utilizes a delicate local processor allocation strategy and BATCH extends the notion of batches to coordinate the processor allocations. Both algorithms use an extended LIST strategy with failure-handling ability to schedule the jobs. On the theoretical side, we derived new approximation results for both algorithms under several classical speedup models. In particular, LPA is shown to be a constant approximation for the roofline model, the communication model, the Amdahl's model, as well as a mix model. We also derived its approximation ratios for the power model and general monotonic model. On the other hand, BATCH achieves $\Theta(\log_2 f_{\max})$-approximation for arbitrary speedup models, where $f_{\max}$ is the maximum number of failures of any job in a failure scenario. All of these results are worst-case results: they hold for any failure scenario. We also derived an $\omega(1)$ lower bound on the average-case performance of BATCH. Extensive simulations show good performance of the two proposed algorithms compared to some baseline heuristics, demonstrating their practical usefulness and robustness under common job speedups and parameter settings.

Future work will be devoted to the investigation of alternative failure models, such as fail-stop errors (as opposed to silent errors) or schedule-dependent failure probabilities (that depend on the number of processors allocated to a job, and hence on its area). One may also consider checkpointing and rollback recovery for long-running jobs to avoid re-executing a failed job from scratch. On the practical side, we seek to validate the performance of our algorithms by evaluating them using datasets extracted from job execution logs with realistic speedup profiles and failure traces.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and suggestions, which greatly helped improve the final version of the paper.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS'67*, pages 483–485, 1967.

[2] K. P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *ICPP*, pages 72–75, 1990.

[3] K. P. Belkhale, P. Banerjee, and W. S. Av. A scheduling algorithm for parallelizable dependent tasks. In *IPPS*, pages 500–506, 1991.

[4] A. Benoit, V. Le Fèvre, L. Perotin, P. Raghavan, Y. Robert, and H. Sun. Resilient scheduling of moldable jobs on failure-prone platforms. In *IEEE Cluster*, 2020.

[5] A. Benoit, V. Le Fèvre, P. Raghavan, Y. Robert, and H. Sun. Resilient scheduling heuristics for rigid parallel jobs. *IJNC*, 11(1), 2021.

[6] J. Blazewicz, M. Machowiak, G. Mounié, and D. Trystram. Approximation algorithms for scheduling independent malleable tasks. In *Euro-Par*, pages 191–197, 2001.

[7] C. Chen, G. Eisenhauer, M. Wolf, and S. Pande. LADR: Low-cost application-level detector for reducing silent output corruptions. In *HPDC*, pages 156–167, 2018.

[8] C.-Y. Chen and C.-P. Chu. A 3.42-approximation algorithm for scheduling malleable tasks under precedence constraints. *IEEE Trans. Parallel Distrib. Syst.*, 24(8):1479–1488, 2013.

[9] M. Chrobak and C. Kenyon-Mathieu. Sigact news online algorithms column 10: Competitiveness via doubling. *SIGACT News*, 37(4):115–126, 2006.

[10] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Discret. Math.*, 2(4):473–487, 1989.

[11] R. A. Dutton and W. Mao. Online scheduling of malleable parallel jobs. In *PDCS*, pages 136–141, 2007.

[12] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[13] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng. Optimal online scheduling of parallel jobs with dependencies. *Journal of Combinatorial Optimization*, 1(4):393–411, 1998.

[14] A. Guermouche, L. Marchal, B. Simon, and F. Vivien. Scheduling trees of malleable tasks for sparse linear algebra. In *Euro-Par*, pages 479–490, 2015.

[15] P.-L. Guhur, H. Zhang, T. Peterka, E. Constantinescu, and F. Cappello. Lightweight and accurate silent data corruption detection in ordinary differential equation solvers. In *Euro-Par*, 2016.

[16] J. T. Havill and W. Mao. Competitive online scheduling of perfectly malleable jobs with setup times. *European Journal of Operational Research*, 187:1126–1142, 2008.

[17] T. Herault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag, 2015.

[18] J. L. Hurink and J. J. Paulus. Online algorithm for parallel job scheduling and strip packing. In C. Kaklamanis and M. Skutella, editors, *Approximation and Online Algorithms*, pages 67–74. Springer, 2008.

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

[20] K. Jansen and F. Land. Scheduling monotone moldable jobs in linear time. In *IPDPS*, pages 172–181, 2018.

[21] K. Jansen and R. Thöle. Approximation algorithms for scheduling parallel jobs. *SIAM Journal on Computing*, 39(8):3571–3615, 2010.

[22] K. Jansen and H. Zhang. Scheduling malleable tasks with precedence constraints. In *SPAA*, page 86–95, 2005.

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

[24] N. Kell and J. Havill. Improved upper bounds for online malleable job scheduling. *J. of Scheduling*, 18(4):393–410, 2015.

[25] E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. *SIAM J. Comput.*, 30(1):300–317, Apr. 2000.

[26] R. Lepère, D. Trystram, and G. J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. In *ESA*, pages 146–157, 2001.

[27] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *SODA*, pages 167–176, 1994.

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

[29] R. H. Möhring, A. S. Schulz, and M. Uetz. Approximation in stochastic scheduling: The power of LP-based priority policies. *J. ACM*, 46(6):924–942, 1999.

[30] G. Mounié, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *SPAA*, pages 23–32, 1999.

[31] G. Mounié, C. Rapine, and D. Trystram. A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM J. Comput.*, 37(2):401–412, 2007.

[32] T. 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.

[33] M. Scharbrodt, T. Schickinger, and A. Steger. A new average case analysis for completion time scheduling. *J. ACM*, 53(1):121–146, 2006.

[34] A. Souza and A. Steger. The expected competitive ratio for weighted completion time scheduling. In *STACS*, pages 620–631, 2004.

[35] G. N. Srinivasa Prasanna and B. R. Musicus. The optimal control approach to generalized multiprocessor scheduling. *Algorithmica*, 15(1):17–49, 1996.

[36] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *SPAA*, 1992.

[37] Q. Wang and K. H. Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM J. Comput.*, 21(2):281–294, 1992.

[38] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen. Fault tolerant matrix-matrix multiplication: Correcting soft errors on-line. In *ScalA'11*, pages 25–28, 2011.

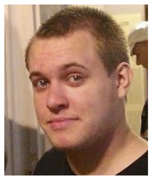
[39] D. Ye, D. Z. Chen, and G. Zhang. Online scheduling of moldable parallel tasks. *J. of Scheduling*, 21(6):647–654, 2018.

[40] D. Ye, X. Han, and G. Zhang. A note on online strip packing. *Journal of Combinatorial Optimization*, 17(4):417–423, 2009.

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

BIOGRAPHIES

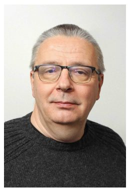
Anne Benoit received the PhD degree from Institut National Polytechnique de Grenoble in 2003, and the Habilitation à Diriger des Recherches (HDR) from ENS Lyon in 2009. She is currently an Associate Professor in the Computer Science Laboratory LIP at ENS Lyon, France, and the IEEE TCPP Chair. She is Associate Editor (in Chief) of Parco, and has been Associate Editor of IEEE TPDS and JPDC. She is a senior member of the IEEE, and she has been elected a Junior Member of Institut Universitaire de France in 2009. Her research interests include multi-criteria scheduling algorithms and resilient techniques for parallel and distributed platforms. See http://graal.ens-lyon.fr/~abenoit/ for further information.



Valentin Le Fèvre is a post-doctoral researcher at Barcelona Supercomputing Center, Spain. He received his PhD in 2020 in the Computer Science Laboratory LIP at Ecole Normale Supérieure de Lyon. He is mainly interested in high-performance computing, in particular resilience and scheduling problems. See http://perso.ens-lyon.fr/valentin.le-fevre/ for further information.

Lucas Perotin is a PhD student in the Computer Science Laboratory LIP at ENS Lyon. He graduated from Ecole Normale Supérieure de Lyon. He is mainly interested in scheduling techniques. See http://perso.ens-lyon.fr/lucas.perotin/ for further information.

Padma Raghavan is Vanderbilt's inaugural Vice Provost for Research and a Professor of Computer Science and Computer Engineering. She joined Vanderbilt in February 2016 from Penn State, where she was the founding Director of the university's Institute for CyberScience. She also served as the Associate Vice President for Research and Strategic Initiatives and as a Distinguished Professor of Computer Science and Engineering at Penn State. She specializes in computational data science and high-performance computing. Her research has been recognized by the NSF CAREER Award (1995), the Maria Goeppert-Mayer Distinguished Scholar Award (2002, University of Chicago and the Argonne National Laboratory), and selection as a Fellow of the Institute of Electrical and Electronic Engineers (IEEE, 2013). See https://engineering.vanderbilt.edu/bio/padma-raghavan/ for further information.



Yves Robert is a Full Professor in the Computer Science Laboratory LIP at ENS Lyon. He is a Fellow of the IEEE and a Senior Member of Institut Universitaire de France. He has been awarded the 2014 IEEE TCSC Award for Excellence in Scalable Computing, the 2016 IEEE TCPP Outstanding Service Award, and the 2020 IEEE CS Charles Babbage Award. He holds a Visiting Scientist position at the Innovative Computing Laboratory at University of Tennessee, Knoxville, since 2011. His main research interests are scheduling techniques, parallel algorithms and resilient approaches for large-scale platforms. See http://graal.ens-lyon.fr/~yrobert/ for further information.

Hongyang Sun is an Assistant Professor in the Department of Electrical Engineering and Computer Science of the University of Kansas, USA. He obtained his Ph.D. in Computer Science from Nanyang Technological University in Singapore, has worked as a Postdoctoral Researcher at ENS Lyon, INRIA (Rhône-Alpes), and IRIT (Toulouse) in France, and held a research faculty position at Vanderbilt University, USA. His research interests include scheduling, resilience and algorithm techniques for high-performance computing, cloud/edge computing, and big-data applications. See https://www.ittc.ku.edu/~sun/ for further information.