Resilient Scheduling of Moldable Jobs on Failure-Prone Platforms

Anne Benoit*, Valentin Le Fèvre*, Lucas Perotin*†, Padma Raghavan†, Yves Robert*‡, Hongyang Sun†
*Laboratoire LIP, ENS Lyon, France
†Vanderbilt University, Nashville, TN, USA
‡University of Tennessee Knoxville, TN, USA

***Abstract*—This paper focuses on 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 of the jobs, or makespan, assuming that jobs are subject to arbitrary failure scenarios, and hence need to be re-executed each time they fail until successful completion. This work generalizes the classical framework where jobs are known offline and do not fail. We introduce a list-based algorithm, and prove new approximation ratios for three prominent speedup models (roofline, communication, Amdahl). We also introduce a batch-based algorithm, where each job is allowed a restricted number of failures per batch, and prove a new approximation ratio for the arbitrary speedup model. 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 list algorithm performs better for the roofline and communication models, while the batch algorithm has better performance for the Amdahl's model. Overall, our best algorithm is within a factor of 1.47 of a lower bound on average over the whole set of experiments, and within a factor of 1.8 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.**

I. Introduction

In the scheduling literature, a moldable job is a parallel job that can be executed on an arbitrary number of processors, but whose execution time depends on the number of processors allotted to it. More precisely, a moldable job allows a variable set of resources for scheduling but requires a fixed set of resources to execute, which the job scheduler must allocate before it starts the job: this corresponds to a variable static resource allocation, as opposed to a fixed static allocation (rigid jobs) and a variable dynamic allocation (malleable jobs) [13]. Moldable jobs can easily adapt to the amount of available resources, contrarily to rigid jobs, while being easy to design and implement, contrarily to malleable jobs. In fact, most computational kernels in scientific libraries are provided as moldable jobs that can be deployed on a wide range of processor numbers[1].

[1]We use *processor* as a generic term for physical resources (cores, nodes, etc).

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, where $1 \leq p \leq P$; here 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 $p+1$-st processor? Several speedup models have been introduced and analyzed, including the roofline model, the communication model, the Amdahl's model, and the (more general) monotonic model, where the area $p \cdot t(p)$ is non-decreasing with p. Section II presents a survey of the most 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) [29], [35]. 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 represent a major challenge on today's HPC platforms [26] 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., [8], [10], [15], [32]). When considering job failures caused by silent errors, we assume the availability of ad-hoc detectors to detect such errors.

Although the primary motivation is to deal with silent errors, this work is agnostic of the type and characteristics of the errors experienced by the jobs. Instead, we focus on a general setting, where we aim at scheduling 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 is half-way 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. More precisely, we aim at designing approximation algorithms that guarantee a makespan within a small factor of the optimal makespan, independently of the job profiles and their failure scenarios.

While scheduling moldable jobs in a failure-free setting is already a difficult problem, this work lays the foundation for the theoretical and practical study of scheduling moldable jobs on failure-prone platforms. The key contributions are the design and analysis of two algorithms (one list-based and one batch-based) with new approximation results for various speedup models. We further show that these algorithms achieve very good performance in practice using an extensive set of simulations. Our main contributions are the following:

- 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 expected performance of an algorithm for general speedup models and under arbitrary failure scenarios.
- We design a list-based scheduling algorithm, and prove new $O(1)$-approximation results for three prominent speedup models, namely the roofline model, the communication model, and the Amdahl's model. For the communication model, our approximation ratio improves upon that of the literature for failure-free jobs.
- We design a batch-based scheduling algorithm, where each job is allowed a restricted number of failures per batch (and is re-executed in the following batch if still not successful). We prove that the algorithm achieves $O(\log_2 f_{\max})$-approximation for the arbitrary speedup model, where $f_{\max}$ denotes the maximum number of failures of any job in a failure scenario.
- We conduct an extensive set of simulations to evaluate and compare different variants of the two proposed algorithms. The results show that they consistently outperform some baseline heuristics. In particular, the list algorithm performs better for the roofline and communication models, while the batch algorithm has better performance for the Amdahl's model. Overall, our best algorithm is within a factor of 1.47 of a lower bound on average and within a factor of 1.8 in the worst case.

The rest of this paper is organized as follows. Section II reviews some related work. The formal model and problem statement are presented in Section III. Section IV describes the two algorithms and analyzes their approximation ratios. Section V presents an extensive set of simulation results and highlights the main findings. Finally, Section VI concludes the paper and discusses future directions.

II. Related Work

In this section, we review some related work on scheduling moldable jobs, and highlight the difference between existing scheduling models and the one considered in this paper.

A. Offline Scheduling of Independent Moldable Jobs

In the offline problem, all jobs are known a priori, along with the execution time $t(p)$ of each job as a function of the processor allocation p. The following surveys some results in the failure-free setting under various job speedup models (the definitions of which can be found in Section III-A).

1) Roofline Model: Some authors have considered this model for moldable jobs with precedence constraints (see Section II-C). We are not aware of any results for independent moldable jobs. In this paper, we present a 2-approximation algorithm for this model when jobs are subject to failures.

2) Communication Model: Havill and Mao [16] presented a shortest execution time (SET) algorithm, which allocates processors to minimize each job's execution time and then schedules it as early as possible. They showed that SET has an approximation ratio around 4. Dutton and Mao [12] 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.

3) Monotonic Model: Belkhale and Banerjee [2] presented a $2/(1 + 1/P)$-approximation algorithm by starting from a sequential schedule and then iteratively incrementing the processor allocations. Błażewicz et al. [6] presented a 2-approximation algorithm while relying on an optimal continuous schedule. Mounié et al. [27] presented a $(\sqrt{3} + \epsilon)$-approximation algorithm with a two-phase approach and dual approximation. Using the same techniques, they later improved the approximation ratio to $1.5 + \epsilon$ [28]. Jansen and Land [20] showed the same $1.5 + \epsilon$ ratio but with a lower runtime complexity as well as a PTAS, when the execution time functions of the jobs admit certain compact encodings.

4) Arbitrary Model: Turek et al. [30] presented a 2-approximation list-based algorithm and a 3-approximation shelf-based algorithm. Ludwig and Tiwari [25] improved the 2-approximation result with lower runtime complexity. When each job 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].

B. Online Scheduling of Independent Moldable Jobs

In an online problem, jobs are released one by one and each released job must be scheduled irrevocably before the next job is revealed. Some algorithms discussed above (e.g., [12],

[16]) make scheduling decisions independently for each job, so their results are directly applicable to this setting with the corresponding competitive ratios. For arbitrary speedup models, Ye et al. [33] presented a general technique to transform any ρ-bounded algorithm[2] for rigid jobs to a 4ρ-competitive algorithm for moldable jobs. Relying on a 6.66-bounded algorithm for rigid jobs [18], [34], they then gave a 26.65-competitive algorithm for moldable jobs. Both algorithms are based on building shelves. They also provided an improved algorithm with a competitive ratio of 16.74 [33].

Our scheduling problem can be considered as semi-online, since all jobs are known offline but their failure scenarios are revealed online. Note that the transformation technique in [33] does not apply here, since it assumes the independence of all jobs, whereas the different executions of a job in our problem (due to failures) have linear dependence.

C. Scheduling Moldable Jobs with Precedence Constraints

Some authors have also studied the problem of scheduling moldable jobs subject to a precedence constraint that is modeled as a directed acyclic graph (DAG).

For the roofline model, Wang and Cheng [31] showed that the earliest completion time (ECT) algorithm is a $(3-2/P)$-approximation. Feldmann et al. [14] proposed an online algorithm for the same model, and showed that it achieves 2.618-competitive even when the job execution times and the DAG structure are unknown. For the more general monotonic model, various constant approximation results have also been obtained (e.g., [3], [7], [9], [21], [22], [24]) under different assumptions on the speedup function and the DAG structures.

In our scheduling problem, 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 corresponding job. However, the failure scenario (thus the complete graph) is not known a priori, which prevents the above algorithms (except the ones in [14], [31]) from being directly applicable, since they all rely on knowing the complete graph in advance.

III. Models

In this section, we introduce the job, speedup and failure models, and formally state the resilient scheduling problem.

A. Job and Speedup Model

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 $J_j \in \mathcal{J}$, let $t_j(p_j)$ denote its execution time when allocated $p_j \in \{1, 2, \dots, P\}$ processors, and the *area* of the job is defined as $a_j(p_j) = p_j \cdot t_j(p_j)$.

[2]An algorithm for rigid jobs is said to be ρ-bounded if its makespan is at most ρ times the lower bound $L = \max\left(\frac{\sum_j t_j p_j}{P}, \max_j t_j\right)$.

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 $\bar{p}_j$, 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 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 $t_j(p_j) = w_j\big(\frac{1-\gamma_j}{p_j} + \gamma_j\big)$, where γ_j denotes the inherently sequential fraction of the job;
- *Arbitrary model*: there are no constraints on $t_j(p_j)$.

B. Failure Model

We consider *silent errors (or SDCs)* that could cause a job to produce erroneous results after an execution attempt. Further, we assume that such errors can be detected using lightweight detectors with negligible overhead at the end of an execution. In that case, the job needs to be re-executed followed by another error detection. This process repeats until the job completes successfully without errors.

Let $\mathbf{f} = (f_1, f_2, \dots, f_n)$ 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.

C. 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 $\mathbf{p} = (\vec{p}_1, \vec{p}_2, \dots, \vec{p}_n)$ of processor allocation vectors for all jobs, where vector $\vec{p}_j = (p_j^{(1)}, p_j^{(2)}, \dots, p_j^{(f_j+1)})$ 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 $\mathbf{s} = (\vec{s}_1, \vec{s}_2, \dots, \vec{s}_n)$ of starting time vectors for all jobs, where vector $\vec{s}_j = (s_j^{(1)}, s_j^{(2)}, \dots, s_j^{(f_j+1)})$ 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 $\mathcal{J}$ under 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 $P \geq 5$ processors [11], the resilient scheduling problem is also $\mathcal{NP}$-complete. We therefore consider approximation algorithms. An algorithm ALG is said to be *r-approximation* if its makespan is at most r times that of an optimal scheduler for any job set $\mathcal{J}$ under any failure scenario $\mathbf{f}$:

$$T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) \leq r \cdot T_{\text{OPT}}(\mathcal{J}, \mathbf{f}, \mathbf{p}^*, \mathbf{s}^*) \ , \tag{2}$$

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 $\mathbf{s}^*$.

D. Expected Makespan

The problem defined 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 a *worst-case* setting.

In contrast, some practical scenarios 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. Then, the probability that the job fails f_j times before succeeding on the $f_j + 1$-st execution is given by $q_j(f_j) = q_j^{f_j}(1 - q_j)$. Assuming that errors occur independently for different jobs, the probability that a failure scenario $\mathbf{f} = (f_1, f_2, \dots, f_n)$ happens can then be computed as $Q(\mathbf{f}) = \prod_{j=1}^{n} q_j(f_j)$. With this probability, we can define the *expected* makespan of an algorithm ALG as $\mathbb{E}(T_{\text{ALG}}) = \sum_{\mathbf{f}} Q(\mathbf{f}) \cdot T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s})$. In this case, an algorithm is said to be r-approximation if its expected makespan satisfies:

$$\mathbb{E}(T_{\text{ALG}}) \leq r \cdot \mathbb{E}(T_{\text{OPT}}) \ , \tag{3}$$

where $\mathbb{E}(T_{\text{OPT}})$ denotes the optimal expected makespan.

For the theoretical analysis (in Section IV), we will focus on bounding the worst-case approximation ratios. For the experimental evaluations (in Section V), we will consider both worst-case and expected ratios as the performance indicators. To instantiate the failure model, we consider silent errors that strike CPUs and registers during the execution of the jobs. In this framework, 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 amount of resources used to execute the job does not matter, even if the parallel execution time depends on the number of allocated processors. Specifically, suppose the occurrence of silent errors follows an exponential distribution with rate λ, then the failure probability for job J_j is a fixed value $q_j = 1 - e^{-\lambda t_j(1)}$, where $t_j(1)$ is the sequential execution time of J_j. Equipped with this model, we report both worse-case and expected performance in Section V, under a variety of experimental scenarios and speedup models.

IV. Resilient Scheduling Algorithms

We present two resilient scheduling algorithms, called LPA-LIST and BATCH-LIST. We derive the approximation ratios of the two algorithms for different speedup models.

A. Makespan Lower Bound

Before presenting the algorithms, we first consider a simple lower bound for the makespan of any scheduling algorithm under a given failure scenario. This generalizes the well-known lower bound [25], [30] 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:

$$t_{\max}(\mathcal{J}, \mathbf{f}, \mathbf{p}) = \max_{1 \leq j \leq n} \sum_{i=1}^{f_j+1} t_j(p_j^{(i)}) \ , \tag{4}$$

$$A(\mathcal{J}, \mathbf{f}, \mathbf{p}) = \sum_{j=1}^{n} \sum_{i=1}^{f_j+1} a_j(p_j^{(i)}) \ . \tag{5}$$

The following shows a lower bound on the makespan of the algorithm for job set $\mathcal{J}$ under failure scenario $\mathbf{f}$ (regardless of its scheduling decision $\mathbf{s}$):

$$\begin{aligned} T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) &\geq L(\mathcal{J}, \mathbf{f}, \mathbf{p}) \\ &= \max\left(t_{\max}(\mathcal{J}, \mathbf{f}, \mathbf{p}), \frac{A(\mathcal{J}, \mathbf{f}, \mathbf{p})}{P}\right) \ . \end{aligned} \tag{6}$$

B. LPA-LIST Scheduling Algorithm

Our first algorithm, called LPA-LIST, adopts a *two-phase* scheduling approach [25], [30]. The first phase uses a *Local Processor Allocation* (LPA) strategy to determine the processor allocation $\mathbf{p}$, and the second phase uses a LIST scheduling strategy to determine the starting time $\mathbf{s}$ of the jobs.

1) LIST Scheduling Strategy: We first discuss the LIST scheduling strategy for the second phase, assuming a given processor allocation $\mathbf{p}$. Thus, this becomes a rigid-job scheduling phase. Algorithm 1 shows the pseudocode of LIST. The strategy first organizes all jobs in a list based on some priority rule. 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. Since the algorithm is triggered whenever a job completes (with errors or successfully), the complexity is $O((f_{\text{sum}}+n)n)$, where $f_{\text{sum}} = \sum_{j=1}^{n} f_j$ denotes the total number of failures from all jobs in a failure scenario. We point out that the algorithm essentially resembles a greedy backfilling strategy.

In our analysis below, we show that the worst-case approximation ratio is independent of the job priorities used in the algorithm, although it may affect the algorithm's practical performance. In Section V, we will consider some commonly used priority rules for the experimental evaluation.

Algorithm 1: LIST (Scheduling Strategy)

```
begin
    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 $J_k$ has errors 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 $P_{avail} \geq p_j^{(f_j+1)}$ 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
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:*

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

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

Proof. We first observe 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. Let $|I_\ell|$ denote the length of interval I_ℓ. The makespan under a failure scenario $\mathbf{f}$ can then be expressed as $T_{\text{LIST}}(\mathbf{f},\mathbf{p},\mathbf{s}) = \sum_{\ell=1}^{v} |I_\ell|$.

Let $p(I_\ell)$ denote the number of utilized processors during an interval I_ℓ. Since the minimum number of utilized processors during the entire schedule is $p_{\min}$, we have $p(I_\ell) \geq p_{\min}$ for all $I_\ell \in \mathcal{I}$. We consider the following two cases:

Case 1: $p_{\min} \geq \frac{P}{2}$. In this case, we have $p(I_\ell) \geq p_{\min} \geq \frac{P}{2}$ for all $I_\ell \in \mathcal{I}$. Based on the definition of total cumulative area, we have $A(\mathbf{f},\mathbf{p}) = \sum_{\ell=1}^{v} |I_\ell| \cdot p(I_\ell) \geq \frac{P}{2} \cdot T_{\text{LIST}}(\mathbf{f},\mathbf{p},\mathbf{s})$. This implies that:

$$T_{\text{LIST}}(\mathbf{f},\mathbf{p},\mathbf{s}) \leq \frac{2A(\mathbf{f},\mathbf{p})}{P} .$$

Case 2: $p_{\min} < \frac{P}{2}$. Let $I_{\min}$ denote the last interval in the schedule with processor utilization $p_{\min}$, and consider a job J_j that is running during interval $I_{\min}$. Necessarily, we have $p_j \leq p_{\min}$. We now 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 running (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 the definition of maximum cumulative execution time, we have $T_1 = \sum_{i=1}^{f_j+1} t_j(p_j^{(i)}) \leq t_{\max}(\mathbf{f},\mathbf{p})$.

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 $p(I) > P - p_{\min}$, since otherwise there are at least $p_{\min} \geq p_j$ available processors during interval I and hence the $i+1$-st 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 $p(I) > P - p_{\min}$, 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 $p(I) > P - p_{\min}$. Based on the definition of total cumulative area, we have $A(\mathbf{f},\mathbf{p}) \geq (P-p_{\min}) \cdot T_2 + p_{\min} \cdot T_1$. The makespan of LIST under failure scenario $\mathbf{f}$ can then be derived as:

$$\begin{aligned} T_{\text{LIST}}(\mathbf{f},\mathbf{p},\mathbf{s}) &= T_1 + T_2 \\ &\leq T_1 + \frac{A(\mathbf{f},\mathbf{p}) - p_{\min} \cdot T_1}{P - p_{\min}} \\ &= \frac{A(\mathbf{f},\mathbf{p})}{P-p_{\min}} + \frac{(P-2p_{\min}) \cdot T_1}{P-p_{\min}} \\ &\leq \frac{A(\mathbf{f},\mathbf{p})}{P-p_{\min}} + \frac{(P-2p_{\min}) \cdot t_{\max}(\mathbf{f},\mathbf{p})}{P-p_{\min}} . \square \end{aligned}$$

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:*

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

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*

$$r(\alpha,\beta) = \begin{cases} 2\alpha, & \text{if } \alpha \geq \beta \\ \frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta, & \text{if } \alpha < \beta \end{cases} \quad (7)$$

Proof. Based on Lemma 1, when $p_{\min} \geq \frac{P}{2}$, we have:

$$T_{\text{LIST}}(\mathbf{f},\mathbf{p},\mathbf{s}) \leq \frac{2A(\mathbf{f},\mathbf{p})}{P} \leq \frac{2\alpha \cdot A(\mathbf{f},\mathbf{p}^*)}{P} \leq 2\alpha \cdot T_{\text{OPT}}(\mathbf{f},\mathbf{p}^*,\mathbf{s}^*).$$

The last inequality above is due to the makespan lower bound, as shown in Inequality (6).

When $p_{\min} < \frac{P}{2}$, we can derive:

$$\begin{aligned}
T_{\textsc{List}}(\mathbf{f},\mathbf{p},\mathbf{s}) &\le \frac{A(\mathbf{f},\mathbf{p})}{P-p_{\min}} + \frac{(P-2p_{\min})\cdot t_{\max}(\mathbf{f},\mathbf{p})}{P-p_{\min}} \\
&\le \frac{\alpha\cdot A(\mathbf{f},\mathbf{p}^*)}{P-p_{\min}} + \frac{\beta(P-2p_{\min})\cdot t_{\max}(\mathbf{f},\mathbf{p}^*)}{P-p_{\min}} \\
&\le \frac{(\alpha+\beta)P-2\beta p_{\min}}{P-p_{\min}}\cdot T_{\textsc{Opt}}(\mathbf{f},\mathbf{p}^*,\mathbf{s}^*) \\
&= \Big(\alpha+\beta+(\alpha-\beta)\frac{p_{\min}}{P-p_{\min}}\Big)\cdot T_{\textsc{Opt}}(\mathbf{f},\mathbf{p}^*,\mathbf{s}^*).
\end{aligned}$$

We have $\frac{1}{P-1} \le \frac{p_{\min}}{P-p_{\min}} < 1$, since $1 \le p_{\min} < \frac{P}{2}$. Therefore, if $\alpha \ge \beta$, we get:

$$T_{\textsc{List}}(\mathbf{f},\mathbf{p},\mathbf{s}) \le 2\alpha\cdot T_{\textsc{Opt}}(\mathbf{f},\mathbf{p}^*,\mathbf{s}^*),$$

and if $\alpha < \beta$, we get:

$$T_{\textsc{List}}(\mathbf{f},\mathbf{p},\mathbf{s}) \le \Big(\frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta\Big)\cdot T_{\textsc{Opt}}(\mathbf{f},\mathbf{p}^*,\mathbf{s}^*).$$

Note that, in this case, $\frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta > 2\alpha$. □

*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 minimum possible execution time and area. Then, it chooses a processor allocation that leads to the smallest ratio $r(\alpha,\beta)$ defined in Equation (7) based on the job's local bounds (α and β) on the area and execution time, which will also hold globally. The algorithm is very simple to implement with complexity $O(nP)$. Once the processor allocation of a job has been decided, the same allocation will be used in the LIST schedule throughout the job's execution until it completes successfully without failures.

C. Performance of LPA-LIST *for Some Speedup Models*

We now analyze the worst-case performance of the LPA-LIST algorithm for moldable jobs that exhibit three prominent speedup models as well as the general monotonic model.

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 $1 \le \bar{p}_j \le P$.

Proposition 1. *The* LPA-LIST *scheduling algorithm is a 2-approximation for jobs with the roofline speedup model.*

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

2) Communication Model: In the communication model [12], [16], the execution time of a job J_j when allocated p processors is given by $t_j(p) = w_j/p + (p-1)c_j$.

Proposition 2. *The* LPA-LIST *scheduling algorithm is a 3-approximation for jobs with the communication model.*

Algorithm 2: LPA (Processor Allocation Strategy)

```
begin
    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 (7);
            if $r(\alpha, \beta) < r_{\min}$ then
                $p_j \leftarrow p$, $r_{\min} \leftarrow r(\alpha, \beta)$;
            end
        end
    end
end
```

Proof sketch. The minimum execution time of a job J_j depends on w_j and c_j, and the minimum area is $a_{\min} = w_j$ (by allocating 1 processor). To prove the proposition, we consider a processor allocation of $p_j \approx \frac{1}{2}\sqrt{\frac{w_j}{c_j}}$ for the job, and discuss several cases. The complete proof is omitted due to space constraint and can be found in the full version [4]. □

We point out that the result of Proposition 2 improves upon the 4-approximation result for the SET algorithm obtained in [16], which is the best ratio known so far for this model. Furthermore, our result extends the one in [16] in two ways: (1) The model in [16] assumes the same communication overhead c for all the jobs, whereas we consider an individual communication overhead c_j for each job J_j; (2) The algorithm in [16] applies to failure-free job executions, whereas our algorithm is able to handle job failures.

3) Amdahl's Model: In the Amdahl's model [1], the execution time of a job J_j when allocated p processors is $t_j(p) = w_j\big(\frac{1-\gamma_j}{p}+\gamma_j\big)$. It is a particular case of the monotonic model (see Section III-A). In the analysis below, we consider an equivalent form of the model: $t_j(p) = \frac{w_j}{p} + d_j$, where w_j denotes the parallelizable work of the job and d_j denotes the inherently sequential work.

Proposition 3. *The* LPA-LIST *scheduling algorithm is a 4-approximation for jobs with the Amdahl's speedup model.*

Proof. The minimum execution time of a job J_j is $t_{\min} = \frac{w_j}{P} + d_j$ (by allocating P processors), and the minimum area is $a_{\min} = w_j + d_j$ (by allocating 1 processor).

We consider a processor allocation of $p_j = \min(\lceil\frac{w_j}{d_j}\rceil, P)$. For the area, we have $a_j(p_j) = w_j + p_j d_j \le w_j + \lceil\frac{w_j}{d_j}\rceil d_j \le w_j + (\frac{w_j}{d_j}+1)d_j = 2w_j + d_j \le 2a_{\min}$. Thus, we get $\alpha = 2$. For the execution time, we consider two cases: (1) If $\lceil\frac{w_j}{d_j}\rceil \le P$, then $p_j = \lceil\frac{w_j}{d_j}\rceil$, and we have $t_j(p_j) = \frac{w_j}{p_j} + d_j \le \frac{w_j}{w_j/d_j} + d_j = 2d_j \le 2t_{\min}$. 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 $2\alpha = 4$. □

While the results above show $O(1)$-approximation of LPA-LIST for three common speedup models, we can show that it is $\Theta(\sqrt{P})$-approximation for the general monotonic model, and the result holds for any algorithm that makes *local* processor allocation decisions based on the individual job characteristics (the proof can be found in [4]). In the next section, we will propose another algorithm that overcomes this limitation by using *coordinated* processor allocations for the jobs.

D. BATCH-LIST *Scheduling Algorithm*

We now present the second algorithm, called BATCH-LIST. Unlike LPA-LIST that 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 a *doubling strategy* for an online scheduling problem [33]. The following describes the BATCH-LIST algorithm in detail.

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 before B_k starts, and let $\mathcal{J}_k = \{J_{k,1}, J_{k,2}, \ldots, J_{k,n_k}\}$ 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; if the job is still not successfully completed after that, it will be handled by the next batch B_{k+1}. Let $\mathbf{f}_k = (f_{k,1}, f_{k,2}, \ldots, f_{k,n_k})$ 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 dependency, 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.

To allocate processors for all sub-jobs (or different execution attempts of all pending jobs) in batch B_k, we adopt a pseudo-polynomial time algorithm, called MT-ALLOTMENT, for series-parallel precedence graphs [24] (of which a set of independent linear chains is a special case). 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 $\vec{p}_{k,j} = (p_{k,j}^{(1)}, p_{k,j}^{(2)}, \ldots, p_{k,j}^{(f_{k,j}+1)})$ be the vector of processor allocations for job $J_{k,j}$, and let $\mathbf{p}_k = (\vec{p}_{k,1}, \vec{p}_{k,2}, \ldots, \vec{p}_{k,n_k})$ 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 arbitrary speedup models.

Lemma 3. *The processor allocation $\mathbf{p}_k$ returned by* MT-ALLOTMENT *for all jobs in batch B_k provides an arbitrarily close approximation to the minimum makespan lower bound as defined in Equation (6), i.e.,*

$$L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}_k) \leq (1+\epsilon) \cdot \min_{\mathbf{p}} L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}) \ . \quad (8)$$

The algorithm has a time complexity polynomial in $1/\epsilon$.

We refer to [24] for a detailed description of the MT-ALLOTMENT algorithm and its analysis[3].

Finally, after deciding the processor allocations, the BATCH-LIST algorithm 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.

E. Performance of BATCH-LIST *for Arbitrary Speedup Model*

We now analyze the 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) $L(\mathcal{J}', \mathbf{f}', \mathbf{p}) \leq L(\mathcal{J}, \mathbf{f}, \mathbf{p})$.
(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 the* BATCH-LIST *algorithm. 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 $\sum_{k=1}^{b} 2^{k-1} = 2^b - 1$. 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 proposition shows the approximation ratio of BATCH-LIST for jobs with any arbitrary speedup model.

Proposition 4. *The* BATCH-LIST *scheduling algorithm 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. From Lemma 5, the total number of batches for a job set $\mathcal{J}$ with failure scenario $\mathbf{f}$ is $b_{\max} = \lceil \log_2(f_{\max}+2) \rceil$. We further have $(\mathcal{J}_k, \mathbf{f}_k) \subseteq (\mathcal{J}, \mathbf{f})$ for any batch B_k, where $1 \leq k \leq b_{\max}$. Let $\mathbf{f}'_k = (f'_{k,1}, f'_{k,2}, \ldots, f'_{k,n_k})$ 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

[3]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.

in each batch, according to Lemmas 1, 3 and 4, we can bound the execution time of any batch B_k as follows:

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

The makespan of the BATCH-LIST algorithm then satisfies:

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

V. 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.

A. Simulation Setup

1) Evaluated Algorithms: We evaluate our two 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 to that of the following two baseline heuristics:

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

2) Priority Rules: We consider three priority rules that have been shown in [5] to give good performance when (rigid) jobs are scheduled with the LIST strategy (Algorithm 1), 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);
- HPA (Highest Processor Allocation);
- LA (Largest Area).

3) Speedup Models: We generate synthetic moldable jobs that follow three speedup models: roofline, communication and Amdahl. Each job J_j is defined by two parameters: the total work w_j (or sequential execution time), which is drawn uniformly in $[5000, 4000000]$, and another parameter that depends on the speedup model and is generated as follows:

- For the roofline model, the maximum degree of parallelism $\bar{p}_j$ is an integer drawn uniformly in $[100, 4000]$;
- For the communication model, 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 $[1, 2]$.
- For the Amdahl's model, the sequential fraction of the job 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 $[0, 10]$.

4) Failure Distribution: 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 III-D), 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 default number of jobs to be $P = 7500$ and $n = 500$, respectively, but we will vary all of these parameters to evaluate their impact on the performance.

5) Evaluation Methodology: We generate 30 different sets of jobs, and for each set, we generate 100 failure scenarios randomly drawn from the failure distribution described above. We then average the simulated makespans of an algorithm over the 100 failure scenarios to estimate its expected makespan for each job set. The expected makespan is then normalized by an expected lower bound (also averaged over the 100 failure scenarios) to estimate the expected ratio. Lastly, this ratio is averaged over the 30 job sets to compute the final expected performance of the algorithm. We also estimate the worst-case performance of the algorithm by using its largest normalized makespan over all job sets and failure scenarios.

For a given job set $\mathcal{J}$ and a failure scenario $\mathbf{f}$, the makespan lower bound given in Equation (6) 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 quality of the optimal schedule, the practical performance of all 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.

B. 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 1 shows the normalized makespans for the 11 combinations of algorithms and priority rules under the three speedup models with $P = 7500, n = 500$, and $\lambda = 10^{-7}$. Note that, for the MINAREA algorithm, priority rules LA and LPT are equivalent, as the algorithm always allocates one processor to all jobs, so only the results of LPT are reported. As we can see, MINAREA fares poorly in all 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, leading to extremely large makespan. The MINTIME algorithm performs well for the roofline model,

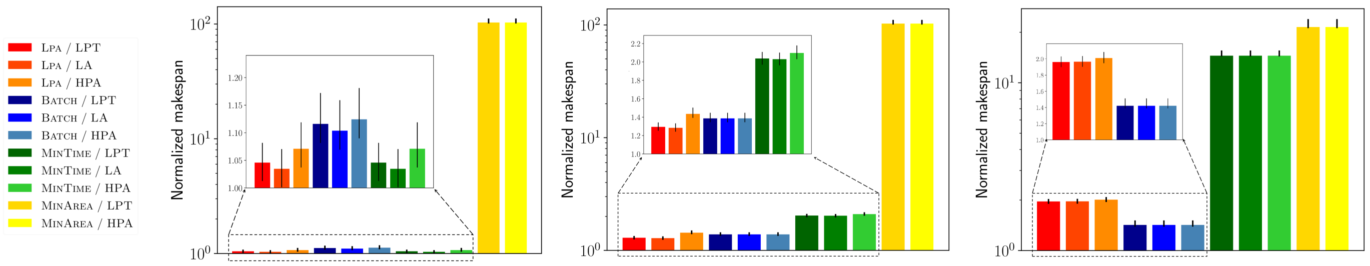

Fig. 1. Performance of all four algorithms for roofline model (left), communication model (middle) and Amdahl's model (right) using different priority rules with $P = 7500$, $n = 500$ and $\lambda = 10^{-7}$. The bars represent expected performance and the endpoints of the lines represent best- and worst-case performance.

but as more overhead is introduced in the communication and Amdahl's models, it continues to allocate a large number of processors to the jobs in order to minimize the execution time. This leads to a significant increase in the total area and hence degrades the performance. On the other hand, the LPA and BATCH algorithms maintain a good balance between the execution time and area of a job, so they perform well for all speedup models, in terms of both expected performance (bars) and worst-case performance (top endpoints of lines). Similar results have also been observed for different combinations of P and n (see [4] for these additional results).

Comparing the three priority rules, no significant difference is observed. In general, LPT and LA perform similarly and slightly better than HPA, which is consistent with the results observed in [5] for scheduling rigid jobs. Given these results, we will omit the MINAREA algorithm altogether (and MINTIME for the Amdahl's model) in the subsequent evaluation, while focusing on comparing the expected performance of the remaining algorithms with the LPT priority rule.

C. *Impact of Different Parameters*

We now study the impact of different parameters (i.e., P, n, and λ) on the performance of the algorithms.

Figure 2 shows the performance when the number of processors P is varied between 1000 and 15000. For the roofline model (left), all three algorithms return the same processor allocation, i.e., the maximum degree of parallelism, for each job. Further, both LPA and MINTIME use the LIST strategy for scheduling, so the two algorithms have exactly the same performance. In contrast, BATCH does not perform as well, because it schedules the jobs in batches, thus needs to wait for every job in a batch to finish before starting the next one, which causes delays. This performance gap also increases with the number of processors. For the communication model (middle), parallelizing a job becomes less efficient due to the extra communication overhead, so BATCH starts to perform better than MINTIME thanks to its better processor allocation strategy. In this model, both BATCH and LPA have similar processor allocations, so the performance difference between the two algorithms is still induced by the idle times at the end of the batches, which again increases with the number of processors. For the Amdahl's model (right), the results look very different, as BATCH now outperforms LPA despite the idle time at the end of each batch. This is due to BATCH's ability to better balance the job execution times globally, which becomes more important for this model. The trend does not seem to be affected by the number of processors.

Figure 3 shows the performance when the number of jobs n is varied between 100 and 1000. We again see the same pattern: BATCH performs the worst in the roofline model, gets better than MINTIME in the communication model, and has the best performance in the Amdahl's model. While the varying number of jobs has a small impact on the performance of LPA and MINTIME, the performance of BATCH improves as the number of jobs increases in the roofline and communication models. Indeed, with a constant number of processors, having more jobs decreases the number of processors per job, thus reduces the performance gap between the algorithms. This is consistent with the results we have observed in Figure 2.

Finally, Figure 4 shows the impact of error rate λ when it is varied between $\lambda = 10^{-8}$ (corresponding to 0.03 error per job on average) and $\lambda = 10^{-6}$ (corresponding to 12 errors per job on average). Once again, the relative performance of the three algorithms remains the same as before under the three speedup models. While the performance of LPA and MINTIME is barely affected, the performance of BATCH gets worse as the error rate λ (and hence the number of failures) increases, which corroborates our theoretical analysis (Proposition 4). In particular, 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 in a large batch, as the actual number of failures may deviate significantly from the anticipated values).

D. *Summary of Results*

Table I summarizes the makespan ratios 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 efficacy of our scheduling algorithms (LPA and BATCH), which outperform the baseline heuristics 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 1.05 on average). For the other two models, the difference between our best algorithm and the baseline is striking! Thanks to its effective processor allocation strategy, LPA achieves very good performance (with a ratio less than 1.4 on average) for the communication model, while MINTIME has a ratio around 2. Results are even more impressive for the Amdahl's model. In

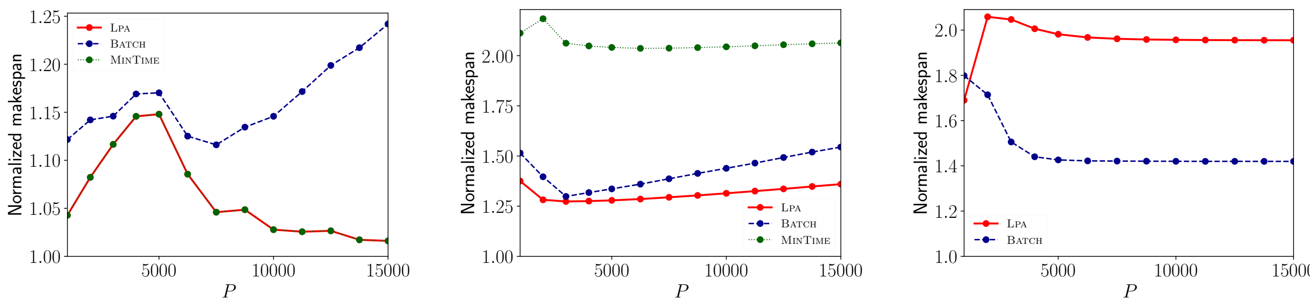

Fig. 2. Performance for roofline model (left), communication model (middle) and Amdahl's model (right) with $n = 500$, $\lambda = 10^{-7}$ and $P \in [1000, 15000]$.

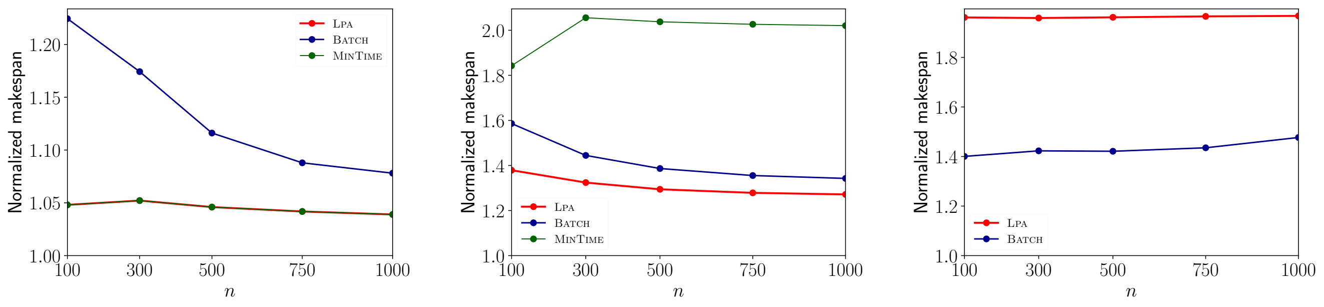

Fig. 3. Performance for roofline model (left), communication model (middle) and Amdahl's model (right) with $P = 7500$, $\lambda = 10^{-7}$ and $n \in [100, 1000]$.

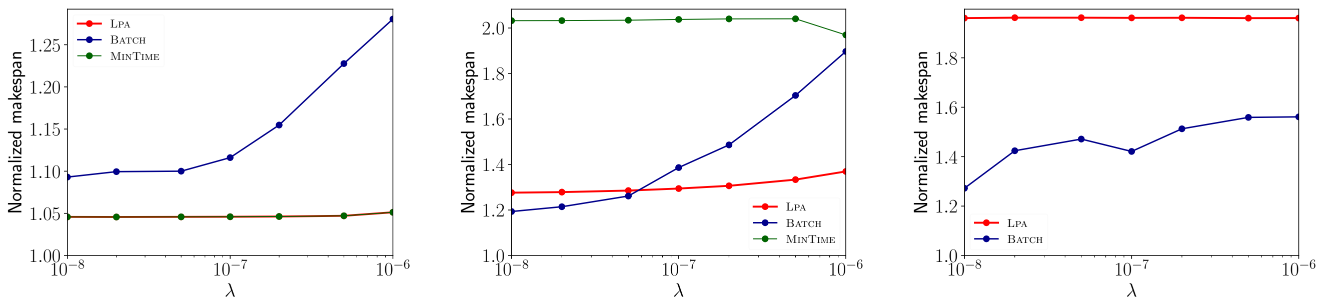

Fig. 4. Performance for roofline model (left), communication model (middle) and Amdahl's model (right) with $P = 7500$, $n = 500$ and $\lambda \in [10^{-8}, 10^{-6}]$.

Table I. Summary of the performance for three algorithms.

Speedup Model		Roofline	Communication	Amdahl
Lpa	Expected	1.055	1.310	1.960
	Maximum	1.148	1.379	2.059
Batch	Expected	1.154	1.430	1.465
	Maximum	1.280	1.897	1.799
MinTime	Expected	1.055	2.040	14.412
	Maximum	1.148	2.184	24.813

this case, Batch has excellent results thanks to its coordinated processor allocation and failure handling, and achieves a ratio around 1.47 on average, against a ratio of more than 14 for MinTime. All of these results are robust with different priority rules and parameter settings.

VI. Conclusion and Future Work

In this paper, we have studied the problem of scheduling moldable parallel jobs on failure-prone platforms. We have presented a formal model of the problem and designed 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 also use an extended List strategy with failure-handling ability to schedule the jobs. On the theoretical side, we have derived new approximation ratios for both algorithms under several job speedup models. Extensive simulation results have also demonstrated their good practical performance against some baseline heuristics and under different parameter settings.

Future work will be devoted to the analysis of average-case performance of the algorithms in addition to their worst-case performance (as was done in this paper). Another direction is to investigate alternative failure models, such as fail-stop errors (as opposed to silent errors as considered in this paper) or schedule-dependent failure probabilities (e.g., the probability that a job fails may depend on the number of processors allocated to it, 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.

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. Scheduling moldable jobs on failure-prone platforms. Research Report RR-9340, INRIA, 2020.

[5] A. Benoit, V. Le Fèvre, P. Raghavan, Y. Robert, and H. Sun. Design and comparison of resilient scheduling heuristics for parallel jobs. In *APDCM*, 2020.

[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. An improved approximation for scheduling malleable tasks with precedence constraints via iterative method. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):1937–1946, 2018.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[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 H. Zhang. Scheduling malleable tasks with precedence constraints. In *SPAA*, page 86–95, 2005.

[22] K. Jansen and H. Zhang. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Trans. Algorithms*, 2(3):416–434, 2006.

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

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

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

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

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

[28] 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.

[29] 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.

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

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

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

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

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

[35] 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.