Scheduling Moldable Jobs on Failure-Prone Platforms

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

Scheduling Moldable Jobs on Failure-Prone Platforms

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

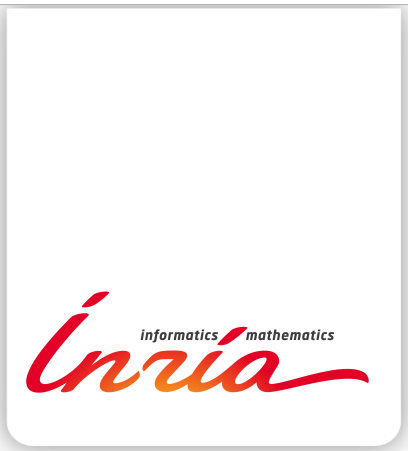
RESEARCH REPORT

N° 9340

May 2020

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9340--FR+ENG



Scheduling Moldable Jobs on Failure-Prone Platforms

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

Project-Team ROMA

Research Report n° 9340 — May 2020 — 37 pages

* LIP, École Normale Supérieure de Lyon, CNRS & Inria, France
† Vanderbilt University, Nashville, TN, USA
‡ University of Tennessee Knoxville, TN, USA

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

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 scheduling objective is to minimize the overall completion time, or makespan, assuming that jobs are subject to arbitrary failure scenarios, and hence may need to be re-executed each time they fail until they complete successfully. 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 only 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, and the results show that they consistently outperform the 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.

Key-words: Resilient scheduling, parallel jobs, moldable jobs, speedup model, failure scenario, transient errors, silent errors, list schedule, batch schedule, approximation ratios.

Ordonnancement avec tolérance aux pannes pour des tâches parallèles à nombre de processeurs programmable

Résumé : Ce rapport étudie l'ordonnancement résilient de tâches sur des plateformes de calcul à haute performance. Dans le problème étudié, il est possible de choisir le nombre constant de processeurs effectuant chaque tâche, en déterminant le temps d'execution de ces dernières selon différent modèles de rendement. Nous décrivons des algorithmes dont l'objectif est de minimiser le temps total d'exécution, sachant que les tâches sont susceptibles d'échouer et de devoir être ré-effectuées à chaque erreur. Ce problème est donc une généralisation du cadre classique où toutes les tâches sont connues à priori et n'échouent pas. Nous décrivons un algorithme d'ordonnancement par listes de priorité, et prouvons de nouvelles bornes d'approximation pour trois modèles de rendement classiques (*roofline, communication, Amdahl*). Nous décrivons également un algorithme d'ordonnancement par lots, au sein desquels les tâches pourront échouer un nombre limité de fois, et prouvons alors de nouvelles bornes d'approximation pour des rendements quelconques. Enfin, nous effectuons des expériences sur un ensemble complet d'exemples pour comparer les niveaux de performance de différentes variantes de nos algorithmes, significativement meilleurs que les algorithmes simples usuels. L'algorithme par listes surpasse l'algorithme par lots sur les modèles *roofline* et *communication*, tandis que l'inverse se produit pour le modèle *Amdahl*. Notre meilleure heuristique est en moyenne à un facteur 1.47 d'une borne inférieure de la solution optimale, et à un facteur 1.8 dans le pire cas.

Mots-clés : Ordonnancement tolérant aux pannes, tâches parallèles, modèles de rendement, nombre de processeurs variable, scénario d'erreurs, erreurs silencieuses, ordonnancement de liste, ordonnancement par paquets, facteurs d'approximation.

1 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 to a variable dynamic allocation (malleable jobs) [12]. 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].

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-decreasing 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 also non-decreasing with p. Section 2 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) [32, 40]. 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 [29] and it will become even more important at

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

exascale [17]. Fortunately, many silent errors can be accurately detected by verifying the integrity of data using dedicated, lightweight detectors (e.g., [7, 9, 15, 37]). 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. Furthermore, we 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 on that of the literature for failure-free jobs.
- We design a batch-based scheduling algorithm, where each job is allowed only a restricted number of failures per batch. We prove that the algorithm achieves a tight $\Theta(\log_2 f_{\max})$-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 list and batch algorithms. The results show that they consistently outperform the 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 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

In this section, we review some related work on scheduling moldable jobs without failures. We consider both offline and online scheduling for independent jobs, as well as scheduling dependent jobs with precedence constraints. We highlight the difference of these scheduling models from the one considered in this paper.

2.1 Offline Scheduling of Independent Moldable Jobs

In the offline scheduling problem, all jobs are known to the scheduler a priori, along with the execution time $t(p)$ of each job as a function of the processor allocation p. Many results have been obtained in the failure-free setting under various job speedup models.

2.1.1 Roofline Model (i.e., $t(p) = w/p$ for $p \leq \bar{p}$ and $t(p) = w/\bar{p}$ for $p > \bar{p}$)

This model assumes linear speedup up to a bounded degree of parallelism $\bar{p}$. Some authors have considered this model for moldable jobs with precedence constraints (see Section 2.3). We are not aware of any results for independent moldable jobs. In this paper, we show that allocating exactly $\bar{p}$ processors to the job and then scheduling all jobs greedily gives a 2-approximation when jobs are subject to failures.

2.1.2 Communication Model (i.e., $t(p) = w/p + (p-1)c$)

This model assumes 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 (around $\sqrt{w/c}$) that minimizes the job's execution time 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 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 [26] presented algorithms with improved approximation ratios for $P \leq 3$ processors.

2.1.3 Monotonic Model (i.e., $t(p) \geq t(p+1)$ and $p \cdot t(p) \leq (p+1) \cdot t(p+1)$)

This 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., $t(p) = w\left(\frac{1-\gamma}{p} + \gamma\right)$ with $\gamma \in (0,1)$, and the power speedup $t(p) = w/p^{\delta}$ [14,34] 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. [5] 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 but with a lower runtime complexity, when the execution time functions of the jobs admit certain compact encodings. They also proposed a PTAS for the problem.

2.1.4 Arbitrary Model

In this model, the execution time $t(p)$ is an unrestricted function of the processor allocation p. With a cost proportional to the total number of processors, this model can be reduced to the monotonic model by discarding those allocations with both larger execution time and area. Turek et al. [35] presented a 2-approximation list-based algorithm and a 3-approximation shelf-based algorithm. Ludwig and Tiwari [28] 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 approx-

imation ratio better than 1.5 unless $P = NP$ [25]. However, when the number of processors is a constant or polynomially bounded by the number of jobs, Jansen et al. [21, 22] showed that a PTAS exists.

2.2 Online Scheduling of Independent Moldable Jobs

In an online scheduling problem, 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 previous section (e.g., [11, 16, 26]) 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 (e.g., [2, 28, 35]) rely on the 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. [38] presented a technique to transform any ρ-bounded algorithm[2] for rigid jobs to a 4ρ-competitive algorithm for moldable jobs. Then, relying on a 6.66-bounded algorithm for rigid jobs [18,39], they 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 [38].

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. [38] 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.

2.3 Scheduling Moldable Jobs with Precedence Constraints

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

Under the roofline model, Wang and Cheng [36] 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

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

processor allocation strategy that minimizes the maximum of critical path length and total area. By adopting a 2-approximation processor allocation technique [33], Lepère et al. [27] presented a 5.236-approximation algorithm under the same model. They also showed that the optimal allocation can be achieved in pseudo-polynomial time for some special graphs, such as series-parallel graphs and trees, thus leading to a 2.618-approximation for these graphs. Jansen and Zhang [24] improved the approximation ratio for general graphs to around 4.73. Recently, Chen [6] developed an iterative method to further improve the ratio, which tends to around 3.42 after a large number of iterations.

When further assuming that the area of a job is a concave function of the number of allocated processors, Jensen and Zhang [23] proposed a 3.29-approximation algorithm via a novel 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 corresponding job. However, the failure scenario (thus the complete graph) is not known a priori, which prevents the above algorithms (except the ones in [13, 36]) from being directly applicable, since they all rely on knowing the complete graph in advance.

3 Models

We formally describe the resilient scheduling problem in this section. We first introduce the job, speedup and failure models in Sections 3.1 and 3.2. Then, we present the problem statement in Section 3.3. Finally, Section 3.4 discusses the expected makespan and average-case analysis.

3.1 Job and Speedup Model

We consider a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of n parallel jobs to be executed on a platform consisting of P identical processors. All jobs are released at the same time, corresponding to the batch scheduling scenario in an HPC environment. 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 $p_j \in \{1, 2, \ldots, P\}$ processors[3], and the *area* of the job is defined as $a_j(p_j) = p_j \cdot t_j(p_j)$.

[3]In this work, we do not allow fractional processor allocation, which could otherwise be realized by timesharing a processor among multiple jobs.

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 $p_j \leq \bar{p}_j$ 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., $t_j(p_j) \geq t_j(p_j + 1)$ and $a_j(p_j) \leq a_j(p_j + 1)$;
- *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 γ_j denotes the inherently sequential fraction of the job;
- *Arbitrary model*: there are no constraints on $t_j(p_j)$.

3.2 Failure Model

We consider silent errors (or SDCs) that could cause a job to produce erroneous results after an execution attempt. 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 $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ 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 the *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:

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

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 it generalizes the failure-free moldable job scheduling problem, which is known to be NP-complete for $P \geq 5$ processors [10], the resilient scheduling problem is also NP-complete. We therefore consider approximation algorithms. A scheduling 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 $\mathbf{s}^*$.

3.4 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 $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ 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 follows:

$$\mathbb{E}(T_{\text{ALG}}) = \sum\nolimits_{\mathbf{f}} Q(\mathbf{f}) \cdot T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) . \tag{3}$$

In this case, an algorithm is said to be r-approximation if its expected makespan satisfies:

$$\mathbb{E}(T_{\text{ALG}}) \leq r \cdot \mathbb{E}(T_{\text{OPT}}) , \tag{4}$$

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

While the approximation ratio of a scheduling algorithm under any failure scenario shows its worst-case performance, the expected ratio may indicate its average-case performance. Clearly, a worst-case ratio will imply the same ratio in the average case, because the inequality is true for each failure scenario (which has a fixed probability), and hence for the weighted sum. However, the converse is not true: an algorithm could have a very good approximation ratio in expectation but performs arbitrarily worse than the optimal in some (low probability) failure scenarios.

For the theoretical analysis (in Section 4), we will focus on bounding the worst-case approximation ratios[4]. For the experimental evaluations (in Section 5), 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 5, under a variety of experimental scenarios and speedup models.

4 Resilient Scheduling Algorithms

In this section, we present two resilient scheduling algorithms, called Lpa-List and Batch-List. We derive the approximation ratios of Lpa-List for three prominent speedup models and of Batch-List for the arbitrary speedup model.

4.1 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 [28,35] 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*

[4]Studying the average-case ratio under either fixed failure probabilities or schedule-dependent failure probabilities will be part of our future work.

area of the jobs under algorithm ALG to be:

$$t_{\max}(\mathcal{J}, \mathbf{f}, \mathbf{p}) = \max_{1 \le j \le n} \sum_{i=1}^{f_j+1} t_j(p_j^{(i)}) \ , \tag{5}$$

$$A(\mathcal{J}, \mathbf{f}, \mathbf{p}) = \sum_{j=1}^{n} \sum_{i=1}^{f_j+1} a_j(p_j^{(i)}) \ . \tag{6}$$

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

$$L(\mathcal{J}, \mathbf{f}, \mathbf{p}) = \max\left(t_{\max}(\mathcal{J}, \mathbf{f}, \mathbf{p}), \frac{A(\mathcal{J}, \mathbf{f}, \mathbf{p})}{P}\right) . \tag{7}$$

Thus, we have:

$$T_{\text{ALG}}(\mathcal{J}, \mathbf{f}, \mathbf{p}, \mathbf{s}) \ge L(\mathcal{J}, \mathbf{f}, \mathbf{p}) \ , \tag{8}$$

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* scheduling approach [28, 35]. The first phase uses a *Local Processor Allocation* (LPA) strategy to determine the processor allocation $\mathbf{p}$ of the jobs, and the second phase uses the LIST scheduling strategy to determine the starting time $\mathbf{s}$ of the jobs.

4.2.1 List Scheduling Strategy

We first discuss the LIST scheduling strategy for the second phase, given a fixed processor allocation $\mathbf{p}$. Since processor allocations have been fixed for all jobs, 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 an existing job J_k completes and hence releases processors, the algorithm checks if job J_k completes with failures. If so, the job will be inserted back into the list, based on its priority, to be re-scheduled later. It then scans the list of pending jobs in sequence and schedules all jobs that can be executed at the current time with the available processors.

In our analysis below, we show that the worst-case approximation ratio of the algorithm is independent of the job priorities, although this choice may affect the algorithm's practical performance. In Section 5, we will consider some commonly used priority rules while experimentally evaluating the performance of the algorithm.

Algorithm 1: LIST (Scheduling Strategy)

```
begin
    Organize all jobs in a list L according to some priority rule;
    P_avail ← P;
    f_j ← 0, ∀j;
    when at time 0 or an existing job J_k completes execution do
        P_avail ← P_avail + p_k^(f_k+1);
        if job J_k failed then
            L.insert_with_priority(J_k);
            f_k ← f_k + 1;
        end
        for j = 1, ..., |L| do
            J_j ← L(j);
            if P_avail ≥ p_j^(f_j+1) then
                execute job J_j at the current time;
                P_avail ← 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:*

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

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 $\mathcal{I} = \{I_1, I_2, \ldots, I_v\}$, 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: $p_{\min} \leq \frac{P}{2}$. 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 $T_1 = (f_j + 1) \cdot t_j(p_j) \leq t_{\max}(\mathbf{f}, \mathbf{p})$.

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:

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

□

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*

$$r(\alpha,\beta) = \begin{cases} 2\alpha, & \text{if } \alpha \geq \beta \\ \frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta, & \text{if } \alpha < \beta \end{cases} \tag{9}$$

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 (7).

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

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

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

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

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

$$T_{\text{List}}(\mathbf{f},\mathbf{p},\mathbf{s}) \leq \left(\frac{P}{P-1}\alpha + \frac{P-2}{P-1}\beta\right) \cdot T_{\text{Opt}}(\mathbf{f},\mathbf{p}^*,\mathbf{s}^*).$$

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

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 as if it were the only job that is present. Algorithm 2 shows its pseudocode. For each job J_j, the strategy first computes its minimum possible execution

Algorithm 2: LPA (Processor Allocation Strategy)

```
begin
    for j = 1, 2, ..., n do
        t_min ← ∞, a_min ← ∞;
        for p = 1, 2, ..., P do
            if t_j(p) < t_min then
                t_min ← t_j(p);
            end
            if p · t_j(p) < a_min then
                a_min ← p · t_j(p);
            end
        end
        p_j ← 0, r_min ← ∞;
        for p = 1, 2, ..., P do
            α ← p · t_j(p)/a_min;
            β ← t_j(p)/t_min;
            compute r(α, β) from Equation (9);
            if r(α, β) < r_min then
                p_j ← p, r_min ← r(α, β);
            end
        end
    end
end
```

time and minimum possible area. Then, it chooses a processor allocation that leads to the smallest ratio $r(\alpha, \beta)$ defined in Equation (9) 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 execution of the job until it completes successfully without failures.

4.3 Performance of Lpa-List for Some Speedup Models

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

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

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

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 $2\alpha = 2$. □

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 $t_j(p) = w_j/p + (p-1)c_j$.

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

Proof. In the communication model, the minimum execution time of a job J_j depends on the values of w_j and c_j, and the minimum area of the job is $a_{\min} = w_j$, achieved by allocating one 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 can be found in the Appendix. □

Remarks. Our result improves upon the 4-approximation for the Set algorithm obtained in [16], which is the best ratio known so far for this model with an arbitrary number of processors. 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.

4.3.3 Amdahl's Model

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

Proposition 3. *The* Lpa-List *scheduling algorithm is a 4-approximation for jobs with the Amdahl's speedup 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 1 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 $2\alpha = 4$. □

4.3.4 Monotonic Model

We now consider the general monotonic model. Recall that a job J_j is *monotonic*, if $t_j(p_1) \geq t_j(p_2)$ and $a_j(p_1) \leq a_j(p_2)$ for any $p_1 \leq p_2$.

Proposition 4. *The* LPA-LIST *scheduling algorithm 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 a processor allocation $p_j = \lfloor \sqrt{P} \rfloor$. Based on the monotonic assumption, we get $a_j(p_j) = p_j t_j(p_j) \leq \sqrt{P} \cdot t_j(1) = \sqrt{P} \cdot a_{\min}$, and $t_j(p_j) \leq \frac{P}{p_j} t_j(P) = O(\sqrt{P}) \cdot t_{\min}$. Thus, based on Lemma 2, we get an approximation ratio of $O(\sqrt{P})$. □

We show that the above approximation ratio is asymptotically tight for any algorithm that makes *local* processor allocation decisions based on the individual job characteristics. Algorithms of this kind include the LPA algorithm considered in this paper and the SET algorithm studied in [16]. In the next section, we will propose another scheduling algorithm that overcomes this limitation by making *coordinated* processor allocation decisions for a set of jobs.

Proposition 5. *Any scheduling algorithm that relies on local processor allocation for each individual job is* $\Omega(\sqrt{P})$*-approximation with the monotonic model.*

Proof. We consider a job with the following execution time characteristic:

$$t(p) = \begin{cases} 1, & \text{if } p \leq \sqrt{P} \\ \frac{\sqrt{P}}{p}, & \text{if } p > \sqrt{P} \end{cases}$$

Therefore, the area of the job satisfies:

$$a(p) = \begin{cases} p, & \text{if } p \leq \sqrt{P} \\ \sqrt{P}, & \text{if } p > \sqrt{P} \end{cases}$$

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 $T_{\text{ALG}} = t(p) = 1$ 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 $T_{\text{ALG}} \geq \frac{n \cdot a(p)}{P} = \sqrt{P}$, and the optimal makespan is $T_{\text{OPT}} = 1$ by allocating 1 processor to each job.

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

4.4 Batch-List Scheduling Algorithm

We now present the second algorithm, called BATCH-LIST. Unlike the LPA-LIST algorithm 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 the *doubling strategy* used in [38] for an online scheduling problem. 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 $\mathcal{J}_k = \{J_{k,1}, J_{k,2}, \ldots, J_{k,n_k}\}$ 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 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 $\mathbf{f}_k = (f_{k,1}, f_{k,2}, \ldots, f_{k,n_k})$ 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 $J_{k,j}^{(1)} \to J_{k,j}^{(2)} \to \cdots \to J_{k,j}^{(g_k)}$ 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 [27] 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 $\vec{p}_{k,j} = (p_{k,j}^{(1)}, p_{k,j}^{(2)}, \ldots, p_{k,j}^{(f_{k,j}+1)})$ 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 any arbitrary speedup model.

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 (7), i.e.,*

$$L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}_k) \leq (1+\epsilon) \cdot \min_{\mathbf{p}} L(\mathcal{J}_k, \mathbf{f}_k, \mathbf{p}) \ . \tag{10}$$

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

We refer to [27] for a detailed description of the MT-ALLOTMENT algorithm and its analysis[5].

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.

4.5 Performance of Batch-List for Arbitrary Speedup Model

We now analyze the performance of the BATCH-LIST algorithm 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 given by $\sum_{k=1}^{b} 2^{k-1} = 2^b - 1$. Thus, if a

[5] 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.

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 6. *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. 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 $\mathbf{f}'_k = (f'_{k,1}, f'_{k,2}, \ldots, f'_{k,n_k})$ 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 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 the Batch-List algorithm satisfies:

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

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

Proposition 7. *The* Batch-List *algorithm is* $\Omega(\log_2(f_{\max}))$*-approximation for jobs with arbitrary speedup model.*

Proof. We consider a set $\mathcal{J} = \{J_1, J_2, \ldots, J_P\}$ of P jobs. For each job J_j, its execution time with one processor is $t_j(1) = 2^{P-j}$, and with increased processor allocation, the execution time decreases linearly until $t_j(P) =$

$2^{P-j}(1-\epsilon)$, where ϵ is a small constant. Further, each job J_j will fails $f_j = 2^{j-1} - 1$ times, thus we have $f_{\max} = f_P = 2^{P-1} - 1$.

To compute the optimal makespan, we could allocate one processor to each job and run the job on the same processor until successful completion. This results in a makespan of 2^{P-1}. Thus, we have $T_{\text{OPT}}(\mathcal{J}, \mathbf{f}) \leq 2^{P-1}$.

Based on Lemma 5, the BATCH-LIST algorithm will complete each job J_j in $\lceil \log_2(f_j + 2) \rceil = j$ batches, and all jobs in $\lceil \log_2(f_{\max} + 2) \rceil = P$ batches. In each batch B_k, where $1 \leq k \leq P$, the set of pending jobs is given by $\mathcal{J}_k = \{J_k, J_{k+1}, \ldots, J_P\}$. For the first batch B_1, it will take at least $2^{P-1}(1-\epsilon)$ time to complete job J_1 and thus the entire batch. For batch B_k, where $1 < k < P$, it will take at least $2^{P-(k+1)}(1-\epsilon)$ time for each execution attempt of job J_{k+1}, which will have 2^{k-1} execution attempts. Thus, batch B_k will take at least $2^{P-2}(1-\epsilon)$ time. The makespan of the BATCH-LIST algorithm for the entire job set $\mathcal{J}$ then satisfies:

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

5 Performance Evaluation

In this section, we evaluate and compare the performance of different scheduling algorithms using simulations on synthetic moldable jobs that follow three speedup models. We first describe the simulation setup in Section 5.1. We then compare the different algorithms and priority rules in Section 5.2. We vary different parameters of our dataset to evaluate their impact on the performance in Section 5.3. Finally, results are summarized in Section 5.4.

5.1 Simulation Setup

5.1.1 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 the BATCH algorithm, 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: it 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: it allocates processors to minimize the area of each job and schedules all jobs using the LIST strategy (Algorithm 1).

5.1.2 Priority Rules

We consider three priority rules that have been shown in [4] 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): a job with a longer processing time will have a higher priority;
- HPA (Highest Processor Allocation): a job with a higher processor allocation will have a higher priority;
- LA (Largest Area): a job with a larger area will have a higher priority.

5.1.3 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 (i.e., the 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 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 $[0, 10]$.

5.1.4 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.

5.1.5 Evaluation Methodology

The evaluation is done as follows: we generate 30 different sets of jobs, and for each of these sets, we generate 100 failure scenarios that are drawn randomly 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. 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.

For a given job set $\mathcal{J}$ and a failure scenario $\mathbf{f}$, the makespan lower bound given in Equation (7) 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 performance of the optimal schedule, the actual 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`.

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 1 shows the normalized makespans for the 11 combinations of algorithms and priority rules under the three speedup models (roofline, communication and Amdahl). Note that, 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 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, which leads to extremely large makespan compared to the best one achievable. The MinTime algorithm performs well for the roofline model, 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

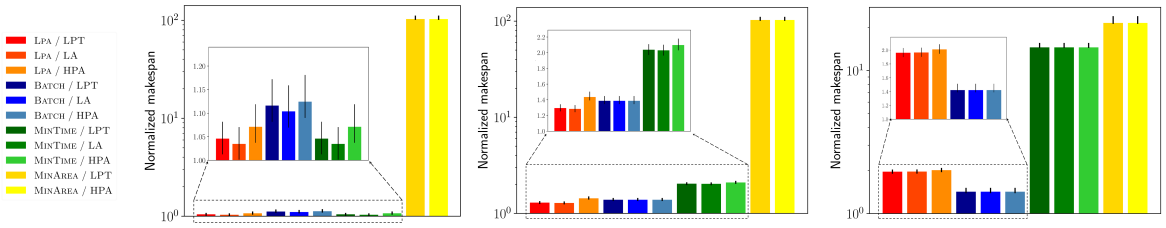


Figure 1: Performance of all four algorithms for roofline model (left), communication model (middle) and Amdahl's model (right) using different priority rules with $P = 7500$, $n = 500$ and $\lambda = 10^{-7}$. The bars represent expected performance and the endpoints of the lines represent best- and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

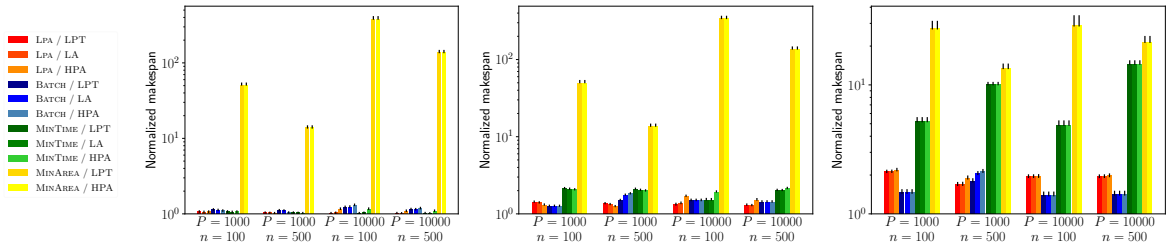


Figure 2: Performance of all four algorithms for roofline model (left), communication model (middle) and Amdahl's model (right) using different priority rules with $\lambda = 10^{-7}$ and four different combinations of P and n.

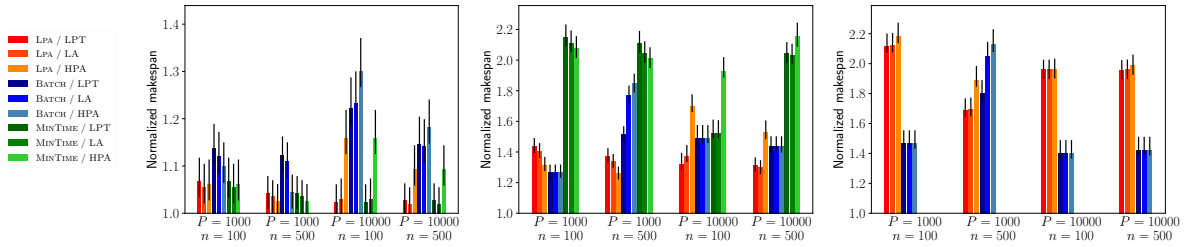


Figure 3: Performance of three algorithms (excluding MinArea) for roofline model (left), communication model (middle) and Amdahl's model (right) using different priority rules with $\lambda = 10^{-7}$ and four different combinations of P and n.

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, thus they perform well for all speedup models, in terms of both expected performance (bars) and worst-case performance (top endpoints of lines).

Figures 2 and 3 further show the results of four other combinations of P and n, and similar performance trends can be observed in these settings. In Figure 3, we notice that the values of P and n have a larger impact on the performance of BATCH in the communication and Amdahl's models, especially at $P = 1000$ and $n = 500$. Indeed, for these two models and 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 therefore closer to the lower bound, and the priority rules do not matter. 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 worse than that of LPA, and the priority rules begin to have an impact.

Comparing the three priority rules, no significant difference is observed. In general, LPT and LA give similar and slightly better results than HPA (except for a small number of processors in the roofline and communication models). This is consistent with the results observed in [4] for scheduling rigid jobs. Given these results, we will omit the MINAREA algorithm altogether (and the MINTIME algorithm for the Amdahl's model) in the subsequent evaluation, while focusing only on comparing the expected performance of the remaining algorithms with the LPT priority rule.

5.3 Impact of Different Parameters

We now study the impact of different parameters, namely, the number of processors P, the number of jobs n, and the error rate λ, on the performance of the algorithms.

Figure 4 shows the performance when the number of processors P is varied between 1000 and 15000 for the three speedup models. 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, and thus needs to wait for every job in a batch to finish before starting the next one, which causes delays. Note that the initial up-and-down of the normalized makespans is due to the upper limit we set on the maximum degree of parallelism (which is 4000):

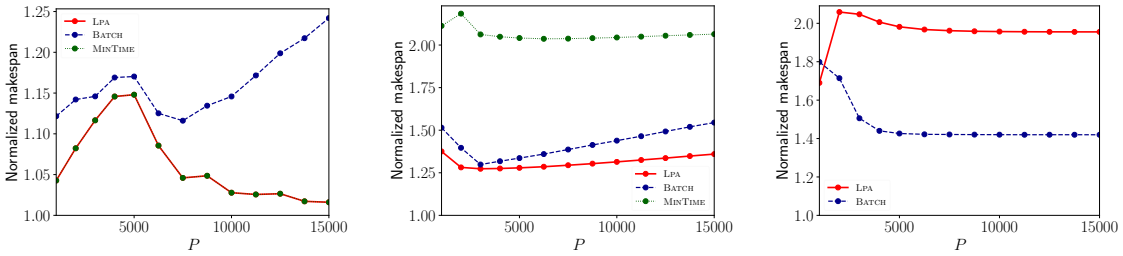


Figure 4: Performance for roofline model (left), communication model (middle) and Amdahl's model (right) with $n = 500$, $\lambda = 10^{-7}$ and $P \in [1000, 15000]$.

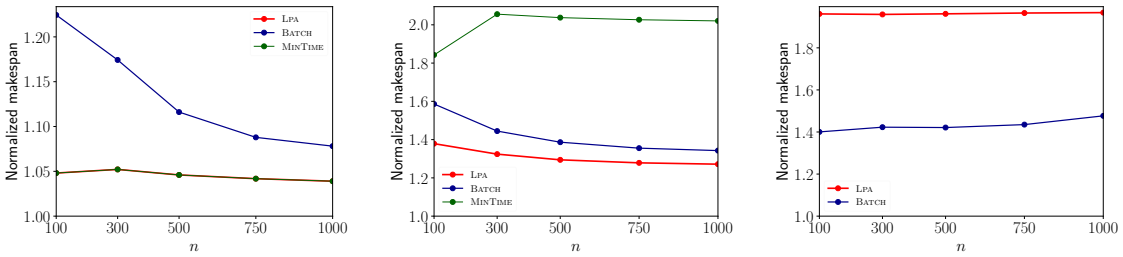


Figure 5: Performance for roofline model (left), communication model (middle) and Amdahl's model (right) with $P = 7500$, $\lambda = 10^{-7}$ and $n \in [100, 1000]$.

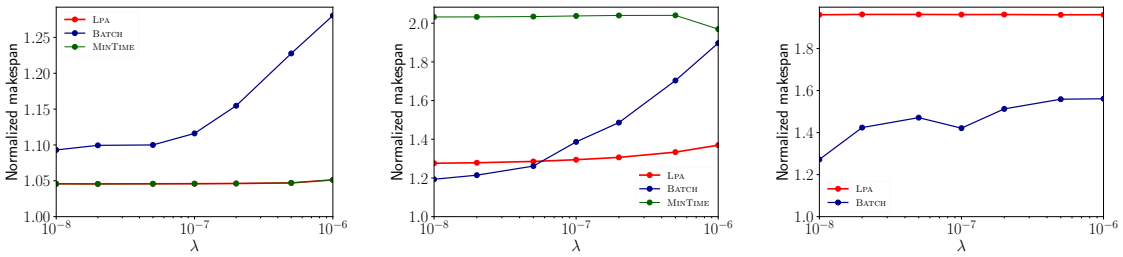


Figure 6: Performance for roofline model (left), communication model (middle) and Amdahl's model (right) with $P = 7500$, $n = 500$ and $\lambda \in [10^{-8}, 10^{-6}]$.

when $P \ll 4000$, very few processors are wasted so the resulting schedules are very efficient; when $P \gg 4000$, most jobs are fully parallelized and thus completed faster. For Batch, however, the proportion of idle processors at the end of a batch also increases with P, which explains the widening of performance gap from the other two algorithms.

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 are again increasing 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 in this case. Moreover, the trend is not affected by the number of processors.

Figure 5 shows the performance when the number of jobs n is varied between 100 and 1000. Again, we can see that 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 P, having more jobs decreases the number of available processors per job, thus reduces the performance gap between scheduling algorithms. This is consistent with the results we have observed in Figure 4.

Finally, Figure 6 shows the impact of the 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 the theoretical analysis (Proposition 6). 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).

Table 1: 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

5.4 Summary of Results

Table 1 summarizes the makespan ratios over the entire set of experiments, in terms of both average-case and worst-case performance. Overall, the results confirm the efficacy of our two resilient 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 roughly around 2. Results are even more impressive for the Amdahl's model. In this case, BATCH achieves excellent results thanks to its coordinated processor allocation and failure handling ability, and achieves a ratio around 1.47 on average, against a ratio of more than 14 for MINTIME. All of these results are shown to be robust with different priority rules and parameter settings.

6 Conclusion and Future Work

In this paper, we have studied the problem of scheduling moldable parallel jobs on failure-prone platforms. Such a job can be allocated with any number of processors before execution, and may be subject to failures. The scheduler must hence re-execute failed jobs while keeping the makespan (overall completion time) as small as possible. 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 and with several classical speedup models. In particular, LPA is a 2-approximation for the roofline model, 3-approximation for the communication model (which improves upon the previously known result of

4-approximation for this model), and a 4-approximation with the Amdahl's model. BATCH achieves $\Theta(\log_2 f_{\max})$-approximation for arbitrary speedup models, where $f_{\max}$ denotes 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.

Extensive simulations demonstrate the good performance of the two resilient scheduling algorithms: LPA is equivalent to the baseline heuristic MINTIME for the roofline model, and achieves an expected makespan very close to the lower bound (around 1.05 times). For the more realistic speedup models, the advantage of our algorithms clearly shows up. LPA has the best performance for the communication model, reducing the expected makespan from around 2 times the lower bound for MINTIME to around 1.4 times. For the Amdahl's model, BATCH becomes the most efficient, with an expected makespan around 1.47 times the lower bound against more than 14 times for MINTIME. Furthermore, the performance of our algorithms is shown to be robust under various priority rules and parameter settings. The results clearly demonstrate the practical usefulness of the proposed strategies, in particular for realistic speedup models.

Future work will be devoted to the study 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). On the practical side, we seek to validate the performance of our algorithms by evaluating them using realistic datasets extracted from job execution logs with 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, P. Raghavan, Y. Robert, and H. Sun. Design and comparison of resilient scheduling heuristics for parallel jobs. In *APDCM*, 2020.

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

[6] 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.

[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] 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.

[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 on-line 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 L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *SODA*, pages 490–498, 1999.

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

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

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

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

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

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

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

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

[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. Skutella. Approximation algorithms for the discrete time-cost tradeoff problem. *Math. Oper. Res.*, 23(4):909–929, 1998.

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

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

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

[37] 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.

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

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

[40] 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.

Appendix. Proof of Proposition 2

In the communication model $t_j(p) = w_j/p+(p-1)c_j$, the minimum execution time $t_{\min}$ of a job J_j depends on the values of w_j and c_j. Let $p_{\min}$ denote the processor allocation that achieves $t_{\min}$, and based on the model, we get that $p_{\min} \approx \sqrt{\frac{w_j}{c_j}}$. The minimum area of the job is $a_{\min} = w_j$, achieved by allocating one processor.

We consider a processor allocation of $p_j \approx \frac{1}{2}\sqrt{\frac{w_j}{c_j}}$ for the job, and show that it achieves the bounds $\alpha = \frac{3}{2}$ and $\beta = \frac{3}{2}$, 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 $2\alpha = 3$. We discuss several cases.

Case 1: If $\frac{1}{2}\sqrt{\frac{w_j}{c_j}} > P$, we set $p_j = P$.

Note that we also have $p_{\min} = P$ in this case, since $t_j(p) = w_j/p+(p-1)c_j$ is a strictly decreasing function of p in $[1, P]$. Thus, we have $t_j(p_j) = \frac{w_j}{P}+(P-1)c_j = t_{\min}$. The area of the job satisfies $a_j(p_j) = w_j+P(P-1)c_j \leq w_j + P^2c_j < w_j + \frac{1}{4}w_j = \frac{5}{4}a_{\min}$.

Case 2: If $\frac{1}{2}\sqrt{\frac{w_j}{c_j}} < \frac{3}{2}$, then the minimum of $t_j(p)$ is achieved at $p^* = \sqrt{\frac{w_j}{c_j}} \in (0,3)$. We consider three subcases depending on the values of p^* and/or $p_{\min}$.

Case 2.1: If $p_{\min} = 1$, we set $p_j = 1$.

In this case, we must have $p^* \in (0,2]$. Therefore, $t_j(p_j) = t_{\min}$ and $a_j(p_j) = a_{\min}$.

Case 2.2: If $p^* \in (1,2]$ and $p_{\min} = 2$, we set $p_j = 1$.

In this case, since $p^* = \sqrt{\frac{w_j}{c_j}} \leq 2$, we have $c_j \geq \frac{1}{4}w_j$. The area of the job using $p_j = 1$ is $a_j(p_j) = a_{\min}$. The minimum execution time of the job is $t_{\min} = \frac{w_j}{2} + c_j \geq \frac{3}{4}w_j$, and the execution time using $p_j = 1$ satisfies $t_j(p_j) = w_j \leq \frac{4}{3}t_{\min}$.

Case 2.3: If $p^* \in (2,3)$, we set $p_j = 2$.

In this case, since $p^* = \sqrt{\frac{w_j}{c_j}} > 2$, we have $c_j < \frac{1}{4}w_j$. Also, we must have $p_{\min} = 2$ or 3. The area of the job using $p_j = 2$ is $a_j(p_j) = w_j + p_j(p_j-1)c_j = w_j + 2c_j < \frac{3}{2}w_j = \frac{3}{2}a_{\min}$. The minimum execution time of the job satisfies $t_{\min} = \frac{w_j}{p_{\min}} + (p_{\min}-1)c_j \geq \frac{w_j}{3} + c_j$, and the execution time using $p_j = 2$ satisfies $t_j(p_j) = \frac{w_j}{2} + c_j \leq \frac{3}{2}(\frac{w_j}{3} + c_j) \leq \frac{3}{2}t_{\min}$.

Case 3: If $\frac{3}{2} \leq \frac{1}{2}\sqrt{\frac{w_j}{c_j}} \leq P$, then we have $c_j \leq \frac{1}{9}w_j$. The minimum of $t_j(p)$ is achieved at $p^* = \sqrt{\frac{w_j}{c_j}}$. Thus, the minimum execution time of the job satisfies $t_{\min} \geq t_j(p^*) = 2\sqrt{w_j c_j} - c_j$.

Let $\frac{1}{2}\sqrt{\frac{w_j}{c_j}} = q + r$, where q denotes the largest integer such that $q \leq \frac{1}{2}\sqrt{\frac{w_j}{c_j}}$ and $r = \frac{1}{2}\sqrt{\frac{w_j}{c_j}} - q$ denotes the remaining fraction. We set p_j by rounding $\frac{1}{2}\sqrt{\frac{w_j}{c_j}}$ as follows:

$$p_j = \begin{cases} q, & \text{if } r \leq 0.2 \\ q+1, & \text{if } r > 0.2 \end{cases} \tag{11}$$

and we consider two subcases depending on the value of p_j.

Case 3.1: If $p_j = q$, we have $\frac{1}{2}\sqrt{\frac{w_j}{c_j}} - 0.2 \leq p_j \leq \frac{1}{2}\sqrt{\frac{w_j}{c_j}}$. The area of the job using p_j is $a_j(p_j) = w_j + p_j(p_j-1)c_j \leq w_j + p_j^2 c_j \leq \frac{5}{4}w_j = \frac{5}{4}a_{\min}$. The execution time of the job using p_j satisfies:

$$\begin{aligned} t_j(p_j) &= \frac{w_j}{p_j} + (p_j - 1)c_j \\ &\leq \frac{w_j}{\frac{1}{2}\sqrt{\frac{w_j}{c_j}} - 0.2} + \Big(\frac{1}{2}\sqrt{\frac{w_j}{c_j}} - 1\Big)c_j \\ &\leq \frac{3}{2}\Big(2\sqrt{w_j c_j} - c_j\Big) \leq \frac{3}{2}t_{\min}\ . \end{aligned}$$

The second last inequality above is shown below:

$$\begin{aligned}
& \frac{w_j}{\frac{1}{2}\sqrt{\frac{w_j}{c_j}} - 0.2} + \Big(\frac{1}{2}\sqrt{\frac{w_j}{c_j}} - 1\Big)c_j \leq \frac{3}{2}\Big(2\sqrt{w_j c_j} - c_j\Big) \\
\Leftarrow \quad & \frac{w_j}{\frac{1}{2}\sqrt{\frac{w_j}{c_j}} - 0.2} \leq \frac{5}{2}\sqrt{w_j c_j} - \frac{1}{2}c_j \\
\Leftarrow \quad & w_j \leq \Big(\frac{5}{2}\sqrt{w_j c_j} - \frac{1}{2}c_j\Big)\Big(\frac{1}{2}\sqrt{\frac{w_j}{c_j}} - 0.2\Big) \\
\Leftarrow \quad & \frac{3}{4}\sqrt{w_j c_j} \leq \frac{1}{4}w_j + \frac{1}{10}c_j \\
\Leftarrow \quad & \frac{3}{4}\sqrt{w_j c_j} \leq \frac{1}{4}w_j \\
\Leftarrow \quad & c_j \leq \frac{1}{9}w_j
\end{aligned}$$

Case 3.2: If $p_j = q + 1$, we have $\frac{1}{2}\sqrt{\frac{w_j}{c_j}} \leq p_j \leq \frac{1}{2}\sqrt{\frac{w_j}{c_j}} + 0.8$. The area of the job using p_j satisfies:

$$\begin{aligned}
a_j(p_j) &= w_j + p_j(p_j - 1)c_j \\
&\leq w_j + \Big(\frac{1}{2}\sqrt{\frac{w_j}{c_j}} + 0.8\Big)\Big(\frac{1}{2}\sqrt{\frac{w_j}{c_j}} - 0.2\Big)c_j \\
&= w_j + \Big(\frac{w_j}{4c_j} + \frac{3}{10}\sqrt{\frac{w_j}{c_j}} - 0.16\Big)c_j \\
&\leq \frac{5}{4}w_j + \frac{3}{10}\sqrt{w_j c_j} \\
&\leq \frac{5}{4}w_j + \frac{1}{10}w_j < \frac{3}{2}a_{\min}\ .
\end{aligned}$$

The second last inequality above is because of $c_j \leq \frac{1}{9}w_j$.

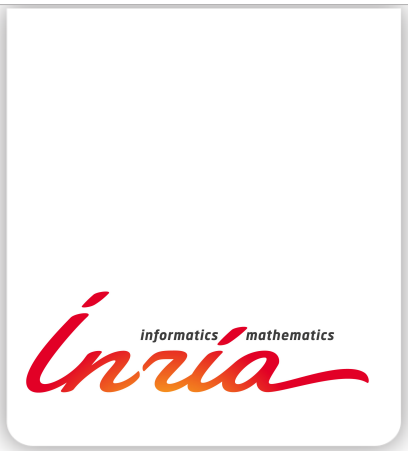
The execution time of the job using p_j satisfies:

$$\begin{aligned}
t_j(p_j) &= \frac{w_j}{p_j} + (p_j - 1)c_j \\
&\leq \frac{w_j}{\frac{1}{2}\sqrt{\frac{w_j}{c_j}}} + \Big(\frac{1}{2}\sqrt{\frac{w_j}{c_j}} + 0.8 - 1\Big)c_j \\
&\leq \frac{5}{2}\sqrt{w_j c_j} - 0.2c_j \\
&\leq \frac{3}{2}\Big(2\sqrt{w_j c_j} - c_j\Big) \leq \frac{3}{2}t_{\min}\ .
\end{aligned}$$

The second last inequality above is shown below:

$$\begin{aligned} & \frac{5}{2}\sqrt{w_j c_j} - 0.2c_j \leq \frac{3}{2}\Big(2\sqrt{w_j c_j} - c_j\Big) \\ \Leftarrow \quad & \frac{13}{10}c_j \leq \frac{1}{2}\sqrt{w_j c_j} \\ \Leftarrow \quad & c_j \leq \Big(\frac{5}{13}\Big)^2 w_j \\ \Leftarrow \quad & c_j \leq \frac{1}{9}w_j \end{aligned}$$

Thus, in all cases, we have shown $a_j(p_j) \leq \frac{3}{2}a_{\min}$ and $t_j(p_j) \leq \frac{3}{2}t_{\min}$. This completes the proof of the proposition.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399