Resilient scheduling on failure-prone platforms

Anne Benoit

LIP, Ecole Normale Supérieure de Lyon, France

Anne.Benoit@ens-lyon.fr

http://graal.ens-lyon.fr/~abenoit/

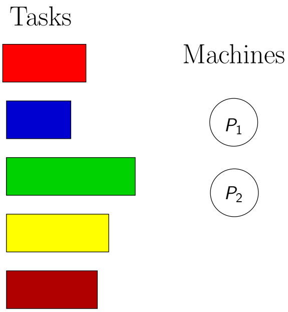
LSV Seminar, December 2020

Motivation

Scheduling: Allocate resources to applications to optimize some performance metrics

- Resources: Large-scale distributed systems with millions of components
- Applications: Parallel applications, expressed as a set of tasks, or divisible application with some work to complete
- Performance metrics: Of course we are concerned with the performance of the applications, but also with resilience and energy consumption

Classical scheduling problems

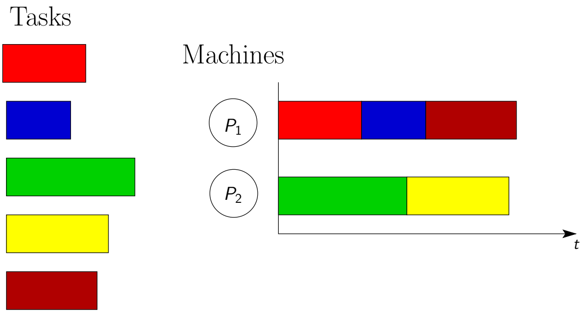


Objectives:

- Minimizing total execution time (C_{max})
- Minimizing weighted sum of execution times $\sum_i w_i C_i$

Results: NP-completeness, algorithms, approximation algorithms, (in-)approximation bounds

Classical scheduling problems

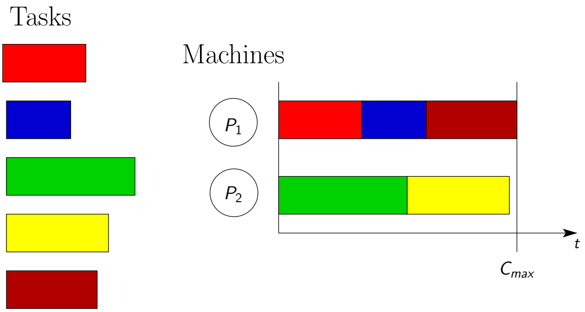


Objectives:

- Minimizing total execution time (C_{max})
- Minimizing weighted sum of execution times $\sum_i w_i C_i$

Results: NP-completeness, algorithms, approximation algorithms, (in-)approximation bounds

Classical scheduling problems

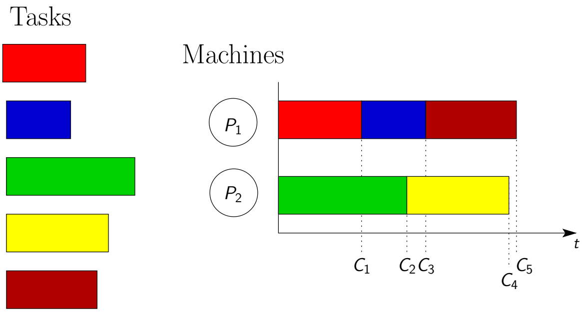


Objectives:

- Minimizing total execution time (C_{max})
- Minimizing weighted sum of execution times $\sum_i w_i C_i$

Results: NP-completeness, algorithms, approximation algorithms, (in-)approximation bounds

Classical scheduling problems

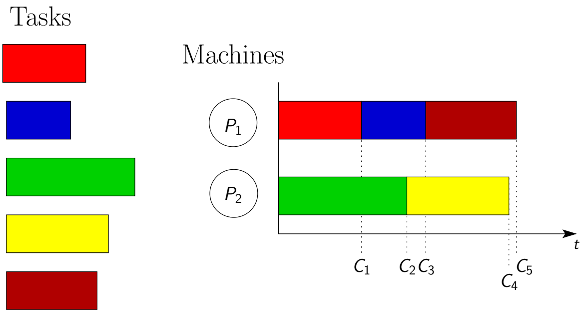


Objectives:

- Minimizing total execution time (C_{max})
- Minimizing weighted sum of execution times $\sum_i w_i C_i$

Results: NP-completeness, algorithms, approximation algorithms, (in-)approximation bounds

Classical scheduling problems



Objectives:

- Minimizing total execution time (C_{max})
- Minimizing weighted sum of execution times $\sum_i w_i C_i$

Results: NP-completeness, algorithms, approximation algorithms, (in-)approximation bounds

Dealing with failures

- Consider one processor (e.g. in your laptop)
  - Mean Time Between Failures (MTBF) = 100 years
  - (Almost) no failures in practice ☺

  Why bother about failures?

- **Theorem:** The MTBF decreases linearly with the number of processors! With 36500 processors:
  - MTBF = 1 day
  - A failure every day on average!

A large simulation can run for weeks, hence it will face failures ☹

Dealing with failures

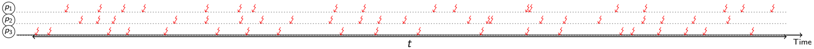
- Consider one processor (e.g. in your laptop)
  - Mean Time Between Failures (MTBF) = 100 years
  - (Almost) no failures in practice ☺

  Why bother about failures?

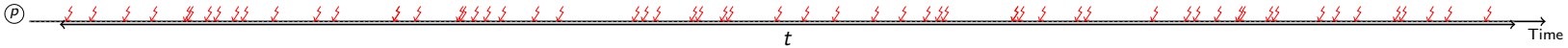
- **Theorem:** The MTBF decreases linearly with the number of processors! With 36500 processors:
  - MTBF = 1 day
  - A failure every day on average!

A large simulation can run for weeks, hence it will face failures ☹

Intuition



If three processors have around 20 faults during a time t ($\mu = \frac{t}{20}$)...

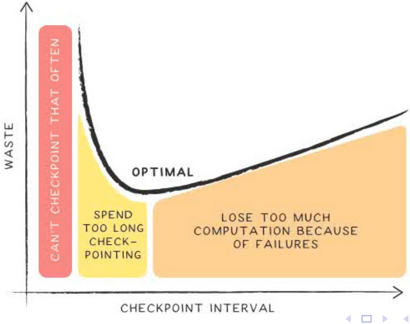


...during the same time, the platform has around 60 faults ($\mu_p = \frac{t}{60}$)

So, how to deal with failures?

Failures usually handled by adding redundancy:

- Re-execute when a failure strikes
- Replicate the work (for instance, use only half of the processors, and the other half is used to redo the same computation)
- Checkpoint the application: Periodically save the state of the application on stable storage, so that we can restart in case of failure without loosing everything



Another crucial issue: Energy consumption

"The internet begins with coal"

- Nowadays: more than 90 billion kilowatt-hours of electricity a year; requires 34 giant (500 megawatt) coal-powered plants, and produces huge CO_2 emissions
- Explosion of artificial intelligence; AI is hungry for processing power! Need to double data centers in next four years
  $\rightarrow$ how to get enough power?
- Failures: Redundant work consumes even more energy

Energy and power awareness $\rightsquigarrow$ crucial for both environmental and economical reasons

Outline

1. Checkpointing for resilience
   - How to cope with errors?
   - Optimization objective and optimal period
   - Optimal period when accounting for energy consumption

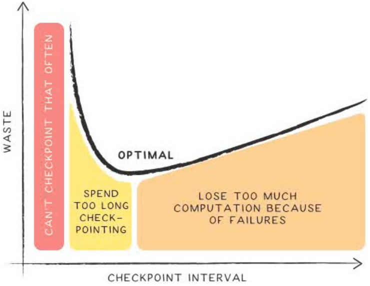
2. Resilient scheduling heuristics for parallel jobs
   - The model
   - Main results for rigid jobs
   - Main results for moldable jobs
   - Simulation results

3. Conclusion

Introduction to resilience

- Fail-stop errors:
  - Component failures (node, network, power, ...)
  - Application fails and data is lost
- Silent data corruptions:
  - Bit flip (Disk, RAM, Cache, Bus, ...)
  - Detection is not immediate, and we may get wrong results

How often should we checkpoint to minimize the waste, i.e., the time lost because of resilience techniques and failures?

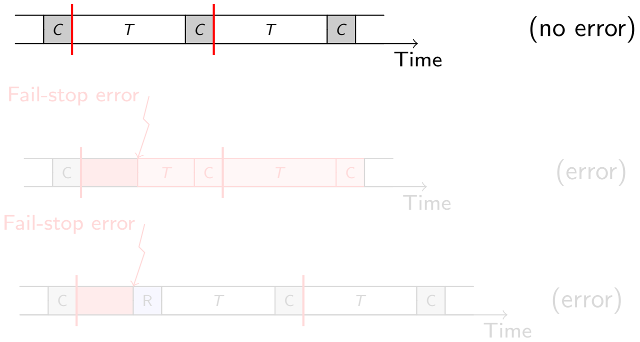


Outline

1. Checkpointing for resilience
   - How to cope with errors?
   - Optimization objective and optimal period
   - Optimal period when accounting for energy consumption
2. Resilient scheduling heuristics for parallel jobs
   - The model
   - Main results for rigid jobs
   - Main results for moldable jobs
   - Simulation results
3. Conclusion

Coping with fail-stop errors

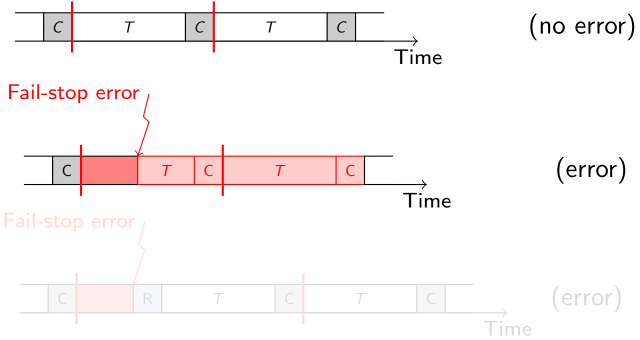
Periodic checkpoint, rollback, and recovery:



- Coordinated checkpointing (the platform is a giant macro-processor)
- Assume instantaneous interruption and detection
- Rollback to last checkpoint and re-execute

Coping with fail-stop errors

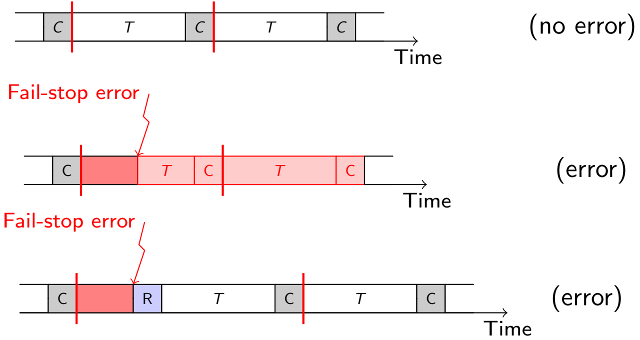
Periodic checkpoint, rollback, and recovery:



- Coordinated checkpointing (the platform is a giant macro-processor)
- Assume instantaneous interruption and detection
- Rollback to last checkpoint and re-execute

Coping with fail-stop errors

Periodic checkpoint, rollback, and recovery:



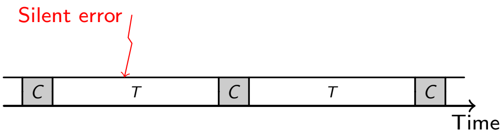
- Coordinated checkpointing (the platform is a giant macro-processor)
- Assume instantaneous interruption and detection
- Rollback to last checkpoint and re-execute

Coping with silent errors

Silent error = detection latency

Error is detected only when corrupted data is activated

Same approach?



Keep multiple checkpoints?

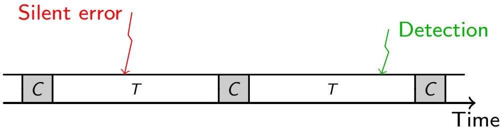
Which checkpoint to recover from?

Need an active method to detect silent errors!

Coping with silent errors

Silent error = detection latency
Error is detected only when corrupted data is activated

Same approach?



Keep multiple checkpoints?

Which checkpoint to recover from?

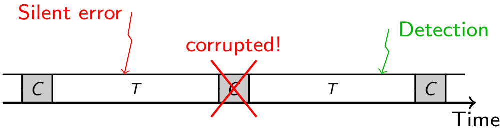
Need an active method to detect silent errors!

Coping with silent errors

Silent error = detection latency

Error is detected only when corrupted data is activated

Same approach?



Keep multiple checkpoints?

Which checkpoint to recover from?

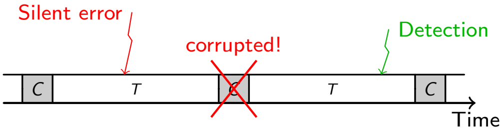
Need an active method to detect silent errors!

Coping with silent errors

Silent error = detection latency

Error is detected only when corrupted data is activated

Same approach?



Keep multiple checkpoints?

Which checkpoint to recover from?

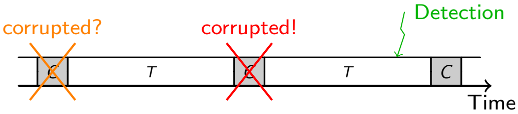
Need an active method to detect silent errors!

Coping with silent errors

Silent error = detection latency

Error is detected only when corrupted data is activated

Same approach?



Keep multiple checkpoints?

Which checkpoint to recover from?

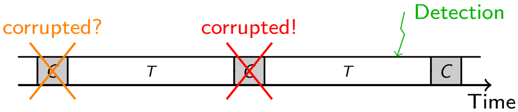
Need an active method to detect silent errors!

Coping with silent errors

Silent error = detection latency

Error is detected only when corrupted data is activated

Same approach?



Keep multiple checkpoints?

Which checkpoint to recover from?

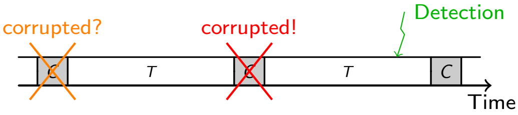
Need an active method to detect silent errors!

Coping with silent errors

Silent error = detection latency

Error is detected only when corrupted data is activated

Same approach?



Keep multiple checkpoints?

Which checkpoint to recover from?

Need an active method to detect silent errors!

Methods for detecting silent errors

General-purpose approaches

- Replication [*Fiala et al. 2012*] or triple modular redundancy and voting [*Lyons and Vanderkulk 1962*]

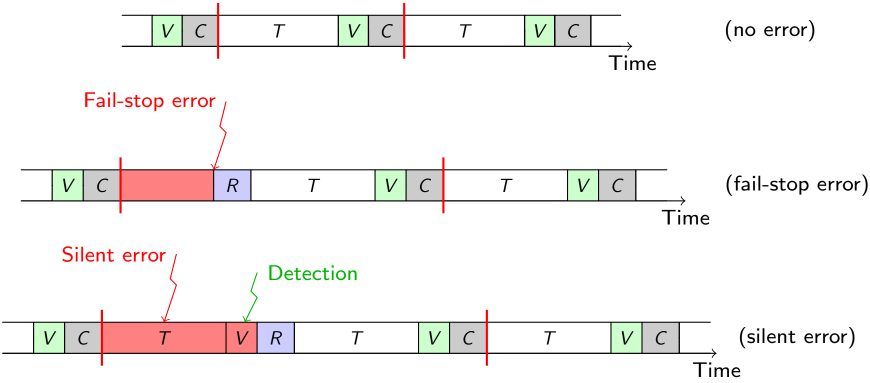
Application-specific approaches

- Algorithm-based fault tolerance (ABFT): checksums in dense matrices Limited to one error detection and/or correction in practice [*Huang and Abraham 1984*]
- Partial differential equations (PDE): use lower-order scheme as verification mechanism [*Benson, Schmit and Schreiber 2014*]
- Generalized minimal residual method (GMRES): inner-outer iterations [*Hoemmen and Heroux 2011*]
- Preconditioned conjugate gradients (PCG): orthogonalization check every k iterations, re-orthogonalization if problem detected [*Sao and Vuduc 2013, Chen 2013*]

Data-analytics approaches

- Dynamic monitoring of HPC datasets based on physical laws (e.g., temperature limit, speed limit) and space or temporal proximity [*Bautista-Gomez and Cappello 2014*]
- Time-series prediction, spatial multivariate interpolation [*Di et al. 2014*]

Coping with fail-stop and silent errors



What is the optimal checkpointing period?

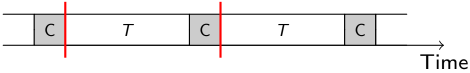
Outline

1 Checkpointing for resilience
- How to cope with errors?
- Optimization objective and optimal period
- Optimal period when accounting for energy consumption

2 Resilient scheduling heuristics for parallel jobs
- The model
- Main results for rigid jobs
- Main results for moldable jobs
- Simulation results

3 Conclusion

Optimization objective (1/2)



- T is the **pattern length** (time without failures)
- C is the checkpoint cost
- $\mathbb{E}(T)$ is the expected execution time of the pattern

By definition, the overhead of the pattern is defined as:

$$\mathbb{H}(T) = \frac{\mathbb{E}(T)}{T} - 1$$

The overhead measures the fraction of **extra time** due to:

- Checkpoints
- Recoveries and re-executions (failures)

The goal is to minimize the quantity: $\mathbb{H}(T)$

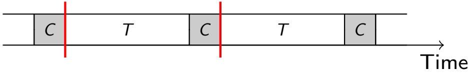
Optimization objective (2/2)

- Goal: Find the **optimal pattern length** T^*, so that the overhead is minimized
- Overhead: $\mathbb{H}(T) = \frac{\mathbb{E}(T)}{T} - 1$

1. Compute expected execution time $\mathbb{E}(T)$ (exact formula)
2. Compute overhead $\mathbb{H}(T)$ (first-order approximation)
3. Derive optimal T^*: fail-stop errors
4. Derive optimal T^*: silent errors
5. Derive optimal T^*: both

1. Expected execution time $\mathbb{E}(T)$

- T: Pattern length
- C: Checkpoint time
- R: Recovery time
- $\lambda^f = \frac{1}{\mu^f}$: Fail-stop error rate



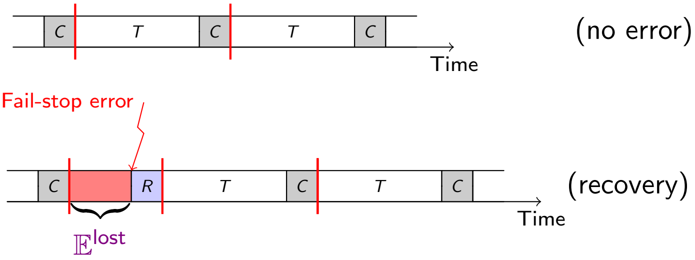
(no error)

$$\mathbb{E}(T) = \mathbb{P}_{no-error}\,(T + C)$$

$$+$$

1. Expected execution time $\mathbb{E}(T)$

- T: Pattern length
- C: Checkpoint time
- R: Recovery time
- $\lambda^f = \frac{1}{\mu^f}$: Fail-stop error rate



$$\mathbb{E}(T) = \mathbb{P}_{no-error}\,(T + C) + \mathbb{P}_{error}\left(\mathbb{E}^{\text{lost}} + R + \mathbb{E}(T)\right)$$

1. Expected execution time $\mathbb{E}(T)$

Assume that failures follow an **exponential distribution** $\text{Exp}(\lambda^f)$

- Independent errors (memoryless property)

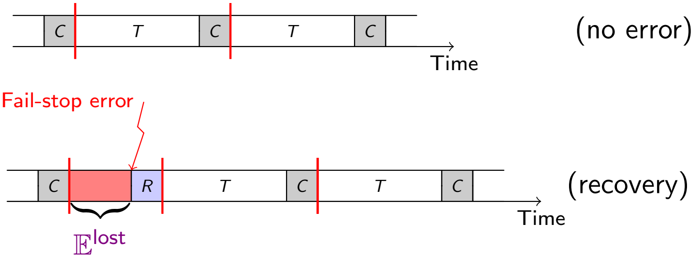
There is at least one error before time t with probability:

$$\mathbb{P}(X \leq t) = 1 - e^{-\lambda^f t} \quad \text{(cdf)}$$

Probability of failure / no-failure

- $\mathbb{P}_{error} = 1 - e^{-\lambda^f T}$
- $\mathbb{P}_{no-error} = e^{-\lambda^f T}$

1. Expected execution time $\mathbb{E}(T)$



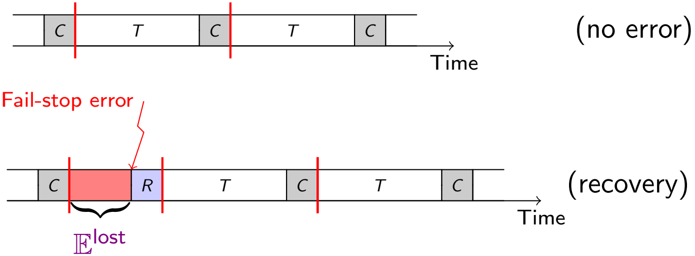
$$\mathbb{E}(T) = e^{-\lambda^f T}(T+C) + (1 - e^{-\lambda^f T})\left(\mathbb{E}^{\text{lost}} + R + \mathbb{E}(T)\right)$$
$$= T + C + (e^{\lambda^f T} - 1)\left(\mathbb{E}^{\text{lost}} + R\right)$$

$\mathbb{E}^{\text{lost}}$ is the time lost when the failure strikes:

$$\mathbb{E}^{\text{lost}} = \int_0^\infty t\mathbb{P}(X = t | X < T)dt = \frac{1}{\lambda^f} - \frac{T}{e^{\lambda^f T} - 1} = \frac{T}{2} + o(\lambda^f T)$$

$\Rightarrow$ **We lose half the pattern upon failure (in expectation)!**

2. Compute overhead $\mathbb{H}(T)$



We use Taylor series to approximate $e^{-\lambda^f T}$ up to first-order terms:

$$e^{-\lambda^f T} = 1 - \lambda^f T + o(\lambda^f T)$$

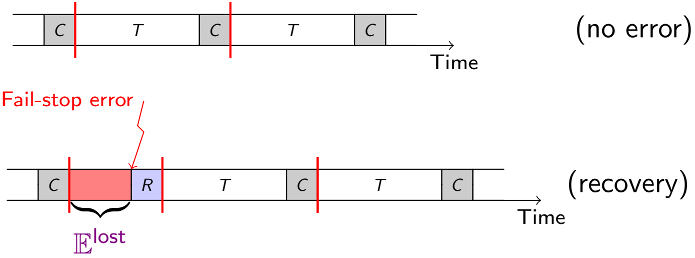
Works well provided that $\lambda^f << T, C, R$

$$\mathbb{E}(T) = T + C + \lambda^f T \left(\frac{T}{2} + R\right) + o(\lambda^f T)$$

Finally, we get the overhead of the pattern:

$$\mathbb{H}(T) = \frac{C}{T} + \lambda^f \frac{T}{2} + o(\lambda^f T)$$

3. Derive optimal T^*: Fail-stop errors



$$\mathbb{H}(T) = \frac{C}{T} + \lambda^f \frac{T}{2} + o(\lambda^f T)$$

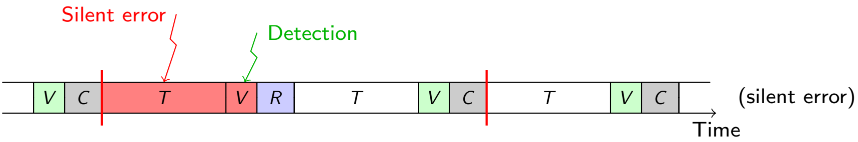
We solve:

$$\frac{\partial \mathbb{H}(T)}{\partial T} = -\frac{C}{T^2} + \frac{\lambda^f}{2} = 0$$

Finally, we retrieve:

$$T^* = \sqrt{\frac{2C}{\lambda^f}} = \sqrt{2\mu^f C}$$

4. Derive optimal T^*: Silent errors



Similar to fail-stop except:

- $\lambda^f \rightarrow \lambda^s$
- $\mathbb{E}^{\text{lost}} = T$
- V: verification time

Using the same approach:

$$\mathbb{H}(T) = \frac{C+V}{T} + \underbrace{\lambda^s T}_{\textit{silent}} + o(\lambda^s T)$$

5. Derive optimal T^*: Both errors

$$\mathbb{H}(T) = \frac{C+V}{T} + \underbrace{\lambda^f \frac{T}{2}}_{fail-stop} + \underbrace{\lambda^s T}_{silent} + o(\lambda T)$$

First-order approximations [Young 1974, Daly 2006, AB et al. 2016]

	Fail-stop errors	Silent errors	Both errors
Pattern	$T+C$	$T+V+C$	$T+V+C$
Optimal T^*	$\sqrt{\frac{C}{\frac{\lambda^f}{2}}}$	$\sqrt{\frac{V+C}{\lambda^s}}$	$\sqrt{\frac{V+C}{\lambda^s+\frac{\lambda^f}{2}}}$
Overhead $\mathbb{H}^*$	$2\sqrt{\frac{\lambda^f}{2}C}$	$2\sqrt{\lambda^s(V+C)}$	$2\sqrt{\left(\lambda^s+\frac{\lambda^f}{2}\right)(V+C)}$

Is this optimal for energy consumption?

5. Derive optimal T^*: Both errors

$$\mathbb{H}(T) = \frac{C+V}{T} + \underbrace{\lambda^f \frac{T}{2}}_{fail-stop} + \underbrace{\lambda^s T}_{silent} + o(\lambda T)$$

First-order approximations [Young 1974, Daly 2006, AB et al. 2016]

	Fail-stop errors	Silent errors	Both errors
Pattern	$T+C$	$T+V+C$	$T+V+C$
Optimal T^*	$\sqrt{\frac{C}{\frac{\lambda^f}{2}}}$	$\sqrt{\frac{V+C}{\lambda^s}}$	$\sqrt{\frac{V+C}{\lambda^s+\frac{\lambda^f}{2}}}$
Overhead $\mathbb{H}^*$	$2\sqrt{\frac{\lambda^f}{2}C}$	$2\sqrt{\lambda^s(V+C)}$	$2\sqrt{\left(\lambda^s+\frac{\lambda^f}{2}\right)(V+C)}$

Is this optimal for energy consumption?

Outline

1. Checkpointing for resilience
   - How to cope with errors?
   - Optimization objective and optimal period
   - Optimal period when accounting for energy consumption
2. Resilient scheduling heuristics for parallel jobs
   - The model
   - Main results for rigid jobs
   - Main results for moldable jobs
   - Simulation results
3. Conclusion

Energy model (1/2)

- Modern processors equipped with dynamic voltage and frequency scaling (DVFS) capability
- Power consumption of processing unit is $P_{idle} + \kappa\sigma^3$, where $\kappa > 0$ and σ is the processing speed
- Error rate: May also depend on processing speed
  - $\lambda(\sigma)$ follows a U-shaped curve
  - increases exponentially with decreased processing speed σ
  - increases also with increased speed because of high temperature

Energy model (2/2)

- Total power consumption depends on:
  - P_{idle}: static power dissipated when platform is on (even idle)
  - $P_{cpu}(\sigma)$: dynamic power spent by operating CPU at speed σ
  - P_{io}: dynamic power spent by I/O transfers (checkpoints and recoveries)
- Computation and verification: power depends upon σ (total time $T_{cpu}(\sigma)$)
- Checkpointing and recovering: I/O transfers (total time T_{io})
- Total energy consumption:

$$Energy(\sigma) = T_{cpu}(\sigma)(P_{idle} + P_{cpu}(\sigma)) + T_{io}(P_{idle} + P_{io})$$

  - Checkpoint: $E^C = C(P_{idle} + P_{io})$
  - Recover: $E^R = R(P_{idle} + P_{io})$
  - Verify at speed σ: $E^V(\sigma) = V(\sigma)(P_{idle} + P_{cpu}(\sigma))$

Bi-criteria problem

Linear combination of execution time and energy consumption:

$$a \cdot Time + b \cdot Energy$$

Theorem

Application subject to both fail-stop and silent errors
Minimize $a \cdot Time + b \cdot Energy$
The optimal checkpointing period is $T^*(\sigma) = \sqrt{\frac{2(V(\sigma)+C_e(\sigma))}{\lambda^f(\sigma)+2\lambda^s(\sigma)}}$,
where $C_e(\sigma) = \frac{a+b(P_{idle}+P_{io})}{a+b(P_{idle}+P_{cpu}(\sigma))} C$

Similar optimal period as without energy, but account for new parameters!

$$T^* = \sqrt{\frac{2(V+C)}{\lambda^f+2\lambda^s}}$$

Outline

1 Checkpointing for resilience
- How to cope with errors?
- Optimization objective and optimal period
- Optimal period when accounting for energy consumption

2 Resilient scheduling heuristics for parallel jobs
- The model
- Main results for rigid jobs
- Main results for moldable jobs
- Simulation results

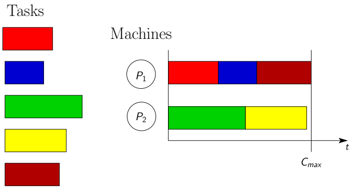
3 Conclusion

Motivation

On large-scale HPC platforms:

- Scheduling parallel jobs is important to improve application performance and system utilization
- Handling job failures is critical as failure/error rates increase dramatically with size of system

We combine job scheduling and failure handling for moldable parallel jobs running on large HPC platforms that are prone to failures



Outline

1 Checkpointing for resilience
- How to cope with errors?
- Optimization objective and optimal period
- Optimal period when accounting for energy consumption

2 Resilient scheduling heuristics for parallel jobs
- The model
- Main results for rigid jobs
- Main results for moldable jobs
- Simulation results

3 Conclusion

Parallel job models

In the scheduling literature:

- **Rigid jobs**: Processor allocation is fixed by the user and cannot be changed by the system (i.e., fixed, static allocation)
- **Moldable jobs**: Processor allocation is decided by the system but cannot be changed once jobs start execution (i.e., fixed, dynamic allocation)
- **Malleable jobs**: Processor allocation can be dynamically changed by the system during runtime (i.e., variable, dynamic allocation)

We focus on moldable jobs, because:

- They can easily adapt to the amount of available resources (contrarily to rigid jobs)
- They are easy to design/implement (contrarily to malleable jobs)
- Many computational kernels in scientific libraries are provided as moldable jobs

Scheduling model

n moldable jobs to be scheduled on P identical processors

- Job j $(1 \leq j \leq n)$: Choose processor allocation p_j $(1 \leq p_j \leq P)$
- Execution time $t_j(p_j)$ of each job j is a function of p_j
- Area is $a_j(p_j) = p_j \times t_j(p_j)$
- Jobs are subject to arbitrary failure scenarios, which are unknown ahead of time (i.e., semi-online)
- Minimize the makespan (successful completion time of all jobs)

Speedup models

- Roofline model: $t_j(p_j) = \frac{w_j}{\max(p_j, \bar{p}_j)}$, for some $1 \leq \bar{p}_j \leq P$
- Communication model: $t_j(p_j) = \frac{w_j}{p_j} + (p_j - 1)c_j$, where c_j is the communication overhead
- Amdahl's model: $t_j(p_j) = w_j\left(\frac{1-\gamma_j}{p_j} + \gamma_j\right)$, where γ_j is the inherently sequential fraction
- Monotonic model: $t_j(p_j) \geq t_j(p_j + 1)$ and $a_j(p_j) \leq a_j(p_j + 1)$, i.e., execution time non-increasing and area is non-decreasing
- Arbitrary model: $t_j(p_j)$ is an arbitrary function of p_j

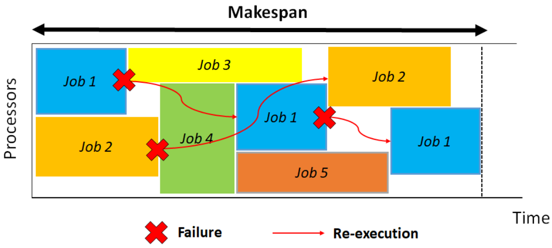
- Rigid jobs: p_j is fixed and hence execution time is t_j

Failure model

- Jobs can fail due to silent errors (or silent data corruptions)
- A lightweight silent error detector (of negligible cost) is available to flag errors at the end of each job's execution
- If a job is hit by silent errors, it must be re-executed (possibly multiple times) till successful completion

A failure scenario $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ describes the number of failures each job experiences during a particular execution

Example: $\mathbf{f} = (2, 1, 0, 0, 0)$ *for an execution of 5 jobs*



Problem complexity

- Scheduling problem clearly NP-hard (failure-free is a special case)
- A scheduling algorithm $\textsc{Alg}$ is said to be a *c-approximation* if its makespan is at most c times that of an optimal scheduler for all possible sets of jobs, and for all possible failure scenarios, i.e.,

$$T_{\textsc{Alg}}(\mathbf{f}, \mathbf{s}) \leq c \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*)$$

- $T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*)$ denotes the optimal makespan with scheduling decision $\mathbf{s}^*$ under failure scenario $\mathbf{f}$

Outline

1. Checkpointing for resilience
   - How to cope with errors?
   - Optimization objective and optimal period
   - Optimal period when accounting for energy consumption

2. Resilient scheduling heuristics for parallel jobs
   - The model
   - Main results for rigid jobs
   - Main results for moldable jobs
   - Simulation results

3. Conclusion

Lower bounds

Rigid jobs: p_j is fixed and job j has execution time t_j

Optimal makespan has two lower bounds:

$$T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \geq t_{\max}(\mathbf{f})$$
$$T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \geq \frac{A(\mathbf{f})}{P}$$

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

List-based algorithm

Resilient list-based scheduling algorithm, and $O(1)$-approximations for any failure scenario:

- Extends classical batch scheduler that combines reservation and backfilling strategies
- Organizes all jobs in a list (or queue) based on some priority rule
- When job J_k completes: processors released; if error, inserted back in the queue; remaining jobs scheduled
- Reservation for first m jobs with highest priorities, at earliest possible time
- Other jobs "backfilled" if reservations not affected
- $m = |Q|$ (Conservative backfilling): reservations for all jobs
- $m = 1$ (Aggressive or EASY backfilling): reservation for 1 job
- $m = 0$ (Greedy scheduler): no reservation

List-based algorithm

Resilient list-based scheduling algorithm, and $O(1)$-approximations for any failure scenario:

- Extends classical batch scheduler that combines reservation and backfilling strategies
- Organizes all jobs in a list (or queue) based on some priority rule
- When job J_k completes: processors released; if error, inserted back in the queue; remaining jobs scheduled
- Reservation for first m jobs with highest priorities, at earliest possible time
- Other jobs "backfilled" if reservations not affected
- $m = |Q|$ (Conservative backfilling): reservations for all jobs
- $m = 1$ (Aggressive or EASY backfilling): reservation for 1 job
- $m = 0$ (Greedy scheduler): no reservation

List-based algorithm: approximation results

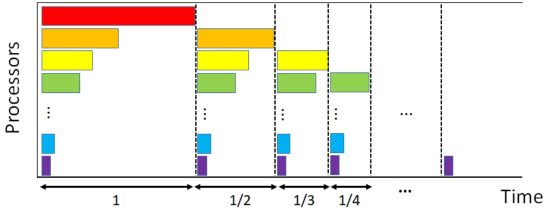
- 2-approximation using Greedy heuristic without reservation
- 3-approximation using Large Job First priority with reservation

The results nicely extend the ones without job failures

[TWY'92: J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. SPAA'92].

Shelf-based algorithm

Resilient shelf-based scheduling heuristic, but $\Omega(\log P)$-approx. for any shelf-based solution in some failure scenario, e.g.:

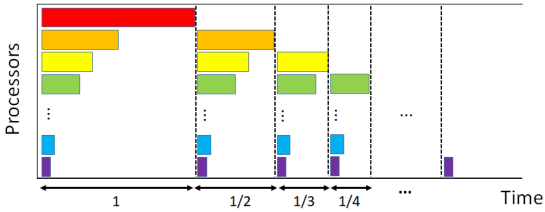


The result defies the $O(1)$-approx. result without failures [TWY'92]

Why not re-execute failed jobs within a same shelf?
Optimal on this example!

Shelf-based algorithm

Resilient shelf-based scheduling heuristic, but $\Omega(\log P)$-approx. for any shelf-based solution in some failure scenario, e.g.:

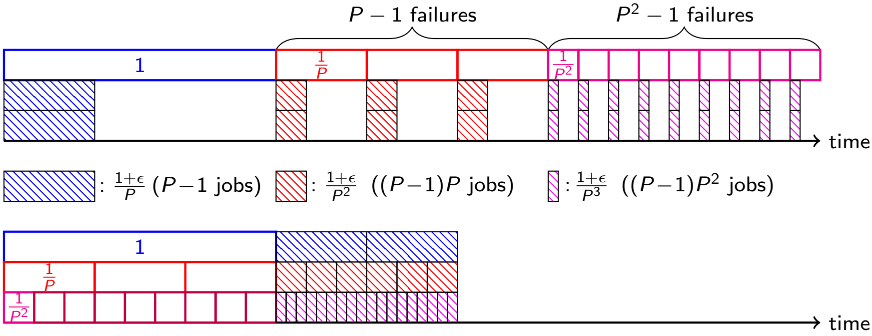


The result defies the $O(1)$-approx. result without failures [TWY'92]

Why not re-execute failed jobs within a same shelf?
Optimal on this example!

Shelf-fill variant: Fill shelfs when error detected

However, there exists a job instance and a failure scenario such that Shelf-fill with the LPT priority rule has an approximation ratio of $\Omega(P)$!



+ *Extensive simulation results of all heuristics using both synthetic jobs and job traces from the Mira supercomputer*

Outline

1. Checkpointing for resilience
   - How to cope with errors?
   - Optimization objective and optimal period
   - Optimal period when accounting for energy consumption
2. Resilient scheduling heuristics for parallel jobs
   - The model
   - Main results for rigid jobs
   - Main results for moldable jobs
   - Simulation results
3. Conclusion

Main results for moldable jobs

Two resilient scheduling algorithms with analysis of approximation ratios and simulation results

1. A list-based scheduling algorithm, called LPA-LIST, and approximation results for several speedup models
2. A batch-based scheduling algorithm, called BATCH-LIST, and approximation result for the arbitrary speedup model
3. Extensive simulations to evaluate and compare (average and worst-case) performance of both algorithms against baseline heuristics

(1) LPA-LIST scheduling algorithm

Two-phase scheduling approach:

- **Phase 1**: Allocate processors to jobs using the Local Processor Allocation (LPA) strategy
    - Minimize a local ratio individually for each job as guided by the property of the LIST scheduling (next slide)
    - The processor allocation p_j will remain unchanged for different execution attempts of the same job j
- **Phase 2**: Schedule jobs with fixed processor allocations using the List Scheduling (LIST) strategy (as in rigid case)
    - Organize all jobs in a list according to any priority order
    - Schedule the jobs one by one at the earliest possible time (with backfilling whenever possible)
    - If a job fails after an execution, insert it back into the queue for rescheduling; Repeat this until the job completes successfully

(1) LPA-LIST scheduling algorithm

Given a processor allocation $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ and a failure scenario $\mathbf{f} = (f_1, f_2, \ldots, f_n)$:

- $A(\mathbf{f}, \mathbf{p}) = \sum_j a_j(p_j)$: total area of all jobs
- $t_{\max}(\mathbf{f}, \mathbf{p}) = \max_j t_j(p_j)$: maximum execution time of any job

Property of LIST Scheduling

For any failure scenario $\mathbf{f}$, if the processor allocation $\mathbf{p}$ 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 $\mathbf{p}^*$ is the processor allocation of an optimal schedule, then a LIST schedule using processor allocation $\mathbf{p}$ is $r(\alpha, \beta)$-approximation:

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

Eq. (1) is used to guide the local processor allocation (LPA) for each job

(1) LPA-LIST scheduling algorithm

Approximation results of LPA-LIST for some speedup models:

Speedup Model	Approximation Ratio
Roofline	2
Communication	3[1]
Amdahl	4
Monotonic	$\Theta(\sqrt{P})$

Advantages and disadvantages of LPA-LIST:

- **Pros**: Simple to implement, and constant approximation for some common speedup models
- **Cons**: Uncoordinated processor allocation, and high approximation for monotonic/arbitrary model

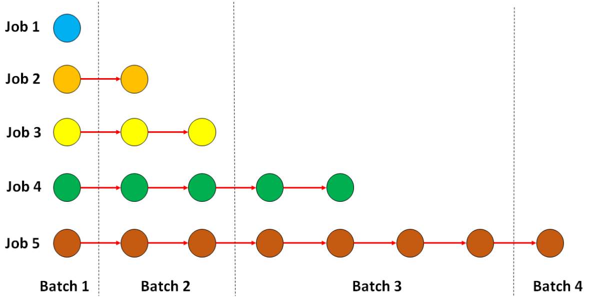
[1] For the communication model, our approx. ratio (3) improves upon the best ratio to date (4), which was obtained without any resilience considerations: [*Havill and Mao. Competitive online scheduling of perfectly malleable jobs with setup times, European Journal of Operational Research, 187:1126–1142, 2008*]

(2) Batch-List scheduling algorithm

Batched scheduling approach:

- Different execution attempts of the jobs are organized in batches that are executed one after another
- In each batch k $(= 1, 2, \dots)$, all pending jobs are executed a maximum of 2^{k-1} times
- Uncompleted jobs in each batch will be processed in the next batch

Example: an execution of 5 jobs under a failure scenario $\mathbf{f} = (0, 1, 2, 4, 7)$



(2) BATCH-LIST scheduling algorithm

Within each batch k:

- Processor allocations are done for pending jobs using the MT-ALLOTMENT algorithm[2], which guarantees near optimal allocation (within a factor of $1 + \epsilon$)
- The maximum of 2^{k-1} execution attempts of the pending jobs are scheduling using the LIST strategy

Approximation Result of BATCH-LIST

The BATCH-LIST algorithm is $\Theta((1 + \epsilon) \log_2(f_{\max}))$-approximation for arbitrary speedup model, where $f_{\max} = \max_j f_j$ is the maximum number of failures of any job in a failure scenario

[2] The algorithm has runtime polynomial in $1/\epsilon$ and works for jobs in SP-graphs/trees (of which a set of independent linear chains is a special case) [*Lepère, Trystram, and Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. European Symposium on Algorithms, 2001*]

Outline

1. Checkpointing for resilience
   - How to cope with errors?
   - Optimization objective and optimal period
   - Optimal period when accounting for energy consumption
2. Resilient scheduling heuristics for parallel jobs
   - The model
   - Main results for rigid jobs
   - Main results for moldable jobs
   - Simulation results
3. Conclusion

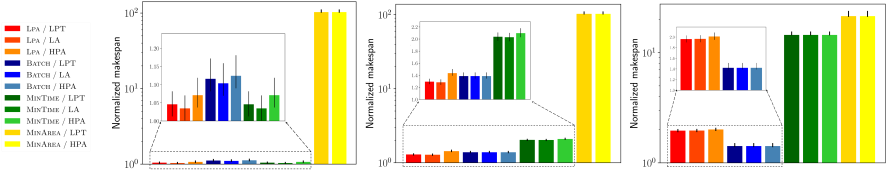
Performance evaluation

We evaluate the performance of our algorithms using simulations

- Synthetic jobs under three speedup models (Roofline, Communication, Amdahl) and different parameter settings
- Job failures follow exponential distribution with varying error rate λ
- Baseline algorithms for comparison:
  - MinTime: allocate processors to minimize execution time of each job and schedule jobs using List
  - MinArea: allocate processors to minimize area of each job and schedule jobs using List
- Priority rules used in List:
  - LPT (Longest Processing Time)
  - HPA (Highest Processor Allocation)
  - LA (Largest Area)

Simulation results — with $P = 7500$, $n = 500$, and $\lambda = 10^{-7}$

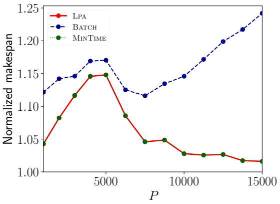
- LPA and BATCH generally perform better than the baselines
- MINTIME performs well for Roofline model, but performs badly for Communication and Amdahl's models
- MINAREA performs the worst for all models
- LPT and LA priorities perform similarly, but better than HPA



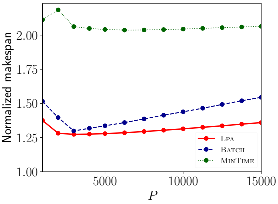
(a) Roofline model (b) Communication model (c) Amdahl's model

Simulation results — with varying number of processors P

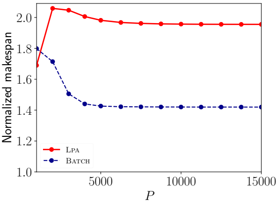
- In Roofline model, LPA (and MINTIME) has better performance, thanks to it simple and effective local processor allocation strategy
- In Communication model, BATCH catches up with LPA and performs better than MINTIME
- In Amdahl's model (where parallelizing a job becomes less efficient due to extra communication overhead), BATCH has the best performance, thanks to its coordinated processor allocation



(d) Roofline model



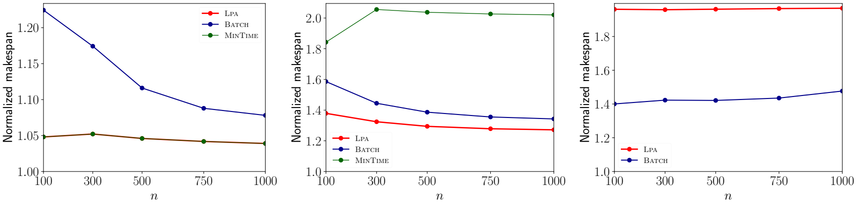
(e) Communication model



(f) Amdahl's model

Simulation results — with varying number of jobs n

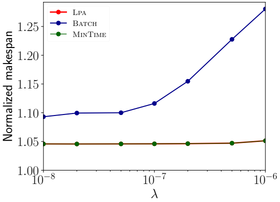
- Same pattern of relative performance (as in last slide) for the three algorithms under the three speedup models
- In Roofline and Communication models, having more jobs reduces number of available processors per job, thus reducing the total idle time between batches ⇒ performance gap between Batch and Lpa is decreasing (instead of increasing as in last slide)



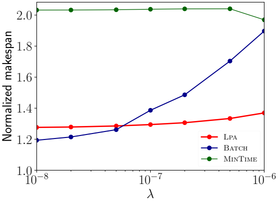
(g) Roofline model (h) Communication model (i) Amdahl's model

Simulation results — with varying error rate λ

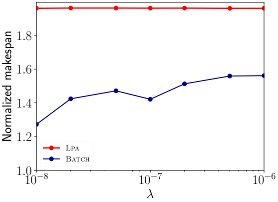
- Same pattern of relative performance (as in last two slides) for the three algorithms under the three speedup models
- A higher error rate increases the number of failures per jobs, which has little impact on LPA and MINTIME, but degrades performance of BATCH (corroborating our approximation results)



(j) Roofline model



(k) Communication model



(l) Amdahl's model

Simulation results — Summary

- Both algorithms (LPA and BATCH) perform significantly better than the baseline (MINTIME and MINAREA)
- Over the whole set of simulations, our best algorithm (LPA or BATCH) is within a factor of 1.47 of the optimal on average, and within a factor of 1.8 of the optimal in the worst case

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

Outline

1. Checkpointing for resilience
   - How to cope with errors?
   - Optimization objective and optimal period
   - Optimal period when accounting for energy consumption
2. Resilient scheduling heuristics for parallel jobs
   - The model
   - Main results for rigid jobs
   - Main results for moldable jobs
   - Simulation results
3. Conclusion

Conclusion

Take-aways:

- Future shared clusters demand simultaneous resource scheduling and resilience considerations for parallel applications
- How to compute the optimal checkpointing period for divisible applications
- Resilient scheduling algorithms for rigid and moldable parallel jobs with provable performance guarantees
- Extensive simulation results demonstrate the good performance of these algorithms under several common speedup models

Future work:

- Analysis of average-case performance of the algorithms (e.g., when some failure scenarios occur with higher probability)
- Considering alternative failure models (e.g., fail-stop errors), and the use of checkpointing to improve efficiency of scheduling
- Performance validation of algorithms using datasets with realistic job speedup profiles and failure traces

Overall: Still a lot of challenges to address, and techniques to be developed for many kinds of high-performance applications, making trade-offs between performance, reliability, and energy consumption

Thanks!!! And a few references:

- A. Benoit, A. Cavelan, Y. Robert, H. Sun. Assessing General-Purpose Algorithms to Cope with Fail-Stop and Silent Errors. TOPC 2016.
- A. Benoit, V. Le Fèvre, P. Raghavan, Y. Robert, H. Sun. Design and Comparison of Resilient Scheduling Heuristics for Parallel Jobs. APDCM 2020.
- A. Benoit, V. Le Fèvre, L. Perotin, P. Raghavan, Y. Robert, H. Sun. Resilient Scheduling of Moldable Jobs on Failure-Prone Platforms. Cluster 2020.