Checkpoint Scheduling

Henri Casanova\textsuperscript{1,2}

\textsuperscript{1}Associate Professor  
Department of Information and Computer Science  
University of Hawai‘i at Manoa, U.S.A.  
\textsuperscript{2}Visiting Associate Professor  
National Institute of Informatics, Japan

NII Seminar Series, October 2013
This is joint work with:

- Marin Bougeret, ENS Lyon, France
- Mickael Rabie, ENS Lyon, France
- Yves Robert, ENS Lyon, France
- Frédéric Vivien, ENS Lyon, France
Yet another twist on "scheduling"

- So far we have talked about scheduling computations and communications on processors and network links
  - the traditional take on scheduling
- In this seminar we’ll talk about something quite different
- Scheduling *checkpoints* on a *failure-prone* parallel platform
- Let’s first provide some context...
Introduction

Towards exascale platforms

- The scale of parallel platforms has been steadily increasing for decades
  - number of cabinets, number of blades, number of processors, number of cores
- Pushed by the need to run ever-larger applications
- Well-known challenges arise (topics of entire conferences/workshops!)
  - Power consumption and heat
  - Network interconnect
  - Resilience to faults
  - Compilers and languages
  - Programming models
  - ...
The scale of parallel platforms has been steadily increasing for decades
- number of cabinets, number of blades, number of processors, number of cores

Pushed by the need to run ever-larger applications

Well-known challenges arise (topics of entire conferences/workshops!)
- Power consumption and heat
- Network interconnect
- Resilience to failures
- Compilers and languages
- Programming models
- ...
Precise definitions of *fault* and *failure* are not often given, and vary between fields.

A *fault* corresponds to an execution that does not go according to specification.

A fault may cause a *failure*, i.e., an unacceptable behavior.

- Some faults can be handled so that they do not become failures (e.g., error correcting codes).

Handling faults in hardware/software: possible but costly

- e.g., redundancy of components.

To build large platforms at acceptable cost, we’re stuck with failure-prone components.
In this seminar we consider a very simple model. We have a set of hosts (e.g., cluster nodes, cores). A host can be hit by a failure: hardware, software, whatever. Failures are i.i.d (independent and identically distributed) and stationary for some probability distribution. All hosts fail the same way. Failures at hosts are independent of each other.

Each resource has a MTBF (Mean Time Between Failure). The HPC literature does the above. The “fault-tolerance literature” studies more complex cases, such as correlated failures.
There are two easy ways to handle failures:

- **Redundancy**
  - Maintain a primary backup: run code on two separate machines
  - Wasteful in terms of resources

- **Checkpoint-Rollback-Recovery**
  - Save "state" occasionally: *checkpointing*
  - When there is a failure, restart the process from the saved state on a *spare host*
  - Wasteful in terms of time (save state even if no failure!)
Two kinds of checkpointing:

- Application-level checkpointing
  - Application code has instructions to save state
  - save only what’s relevant
  - application code must be modified

- System-level checkpointing
  - A process image is saved (registers, stack, heap, etc.)
  - application code is not modified
  - saves a much larger state

Regardless, we assume that there is a way to checkpoint and a way to recover from a checkpoint
Where are checkpoints saved?

- Saved on secondary storage at local host
  - Can only recover on the same host, which may not be possible and could take time
- Saved in memory at another host
  - So-called “diskless” checkpointing
  - Must be sure that other host doesn’t fail
- Saved to some fast and reliable (network-attached) storage system
  - The typical assumption for high-end large machines
**Checkpoint time, recovery time, down time**

- **Checkpoint time:** $C$ seconds
  - Depends on checkpoint size, storage medium
- **Recovery time:** $R$ seconds
  - Depends on checkpoint size, storage medium
- **Downtime:** $D$ seconds
  - After a failure, time for the *logical* host to start a recovery
  - Typically: constant time to bring a spare host on-line

---

### Diagram

- **Failure**
- **Useless checkpoint**
- **Time**
Checkpointing parallel applications

- We assume a tightly-coupled parallel application (all processes must run synchronously)
- Two options for checkpointing
  - **Uncoordinated checkpointing**
    - Processes perform checkpoints at different times
    - 😊 light load on the storage system
    - 😞 recovery is complicated (in-flight messages)
  - **Coordinated checkpointing**
    - All processes perform a checkpoint at the same time
    - 😊 can be done at convenient times (e.g., no pending communication)
    - 😞 possibly heavy load on the storage system
Coordinated checkpointing model
Checkpointing strategy

- When do we take checkpoints?
- The common approach is to checkpoint *periodically*
- The question becomes: *what checkpointing frequency?*
- Trade-off:
  - Not too high, otherwise too much overhead during fault-free execution (too much "green")
  - Not too low, otherwise too much wasted work if fault (too much "blue")
- This is a scheduling problem, albeit quite different from what we’ve seen so far in this seminar series
Why Exponential failures?

- A popular assumption: Poisson failure arrivals
  - Exponentially distributed *inter-arrival times*
- The mean time between failure is $1/\lambda$
- Probability density: $P(x) = \lambda e^{-\lambda x}$
- This distribution is *memoryless*
  - The time to the next failure does not depend on the time since the last failure
- Very convenient for analytical developments, as we’ll see
Periodic checkpointing

- Periodic coordinated checkpointing has been used in practice for decades
- Periodic coordinated checkpointing has been studied in theory for decades
- Many authors have proposed analytical expressions for the “optimal” checkpointing frequency or approximations thereof
- A famous such expression is given by J.W. Young
- Let’s review how this bound is achieved, which will provide context for recent results
Let $T$ be the time between checkpoints (frequency $= 1/T$)
Let $T_w$ be the compute time wasted due to a failure
Let us consider an interval $T_i$ in between two failures

\[ T_i = n \times (T + C) + T_w \]
Objective: minimize \( \mathbb{E}[T_w] \) (\( T_w = T_i - n(T + C) \))

Assumption #1: failure arrival times follow an Exponential distribution of mean \( 1/\lambda \)

Assumption #2: failures do not occur during checkpointing

Assumption #3: failures do not occur during recovery

Assumption #4: failures do not occur during downtime (i.e., on another processor)
Exponential failures

State-of-the-art

Obtaining the approximation

- If $T_i$ is in between $n(T+C)$ and $(n+1)(T+C)$, then
  $$T_w = T_i - nT$$
- subtract from the whole time the "useful" compute time
- Therefore:

$$\mathbb{E}[T_w] = \sum_{n=0}^{\infty} \int_{n(T+C)}^{(n+1)(T+C)} (t - nT) \lambda e^{\lambda t} \, dt$$

$$\Rightarrow \quad \cdots \quad \Rightarrow \quad \mathbb{E}[T_w] = 1/\lambda + T/(1 - e^{\lambda(T+C)})$$

$$\Rightarrow \quad \frac{d\mathbb{E}[T_w]}{dT} = \frac{1 - e^{\lambda(T+C)} + Te^{\lambda(T+C)}}{(1 - e^{\lambda(T+C)})^2}$$
Obtaining the approximation

- To find the optimal $T$ we solve: $e^{\lambda T} e^{\lambda C} (1 - \lambda T) - 1 = 0$
- $\lambda T$ is small (because $1/\lambda$ is large)
- Using the approx $e^{\lambda T} = 1 + \lambda T + \lambda T^2/2$ and neglecting terms of degree 3, we obtain $\frac{1}{2} (\lambda T)^2 = 1 - e^{-\lambda C}$
- $\lambda C$ is even smaller (because $C$ is small), and we use the approx $e^{-\lambda C} = 1 - \lambda C$
- We obtain Young’s approximation:
  $$T_{opt} \sim \sqrt{2C/\lambda}$$
- Example: $MTBF = 48$ hours, $C = 1$ minute, then checkpoint every $T_{opt} = 75.9$ minutes
With recovery time, Young’s approximation becomes
\[ T_{opt} \sim \sqrt{2C(R + 1/\lambda)} \]

J.T. Daly, *A higher order estimate of the optimum checkpoint interval for restart dumps*, FGCS 2006,

Daly goes further and proposes the following approximation:
\[
T_{opt} = \sqrt{2C/\lambda} \left[ 1 + \sqrt{\frac{C\lambda}{18}} + \frac{C\lambda}{18} \right] - C, \quad \text{if} \quad C < \frac{2}{\lambda}
\]
\[
T_{opt} = \frac{1}{\lambda}, \quad \text{if} \quad C \geq \frac{2}{\lambda}
\]

Example: \( MTBF = 48 \) hours, \( C = 1 \) minute, then checkpoint every \( T_{opt} = 75.2 \) minutes
Daly’s bound

- Pretty close to Young’s bound unless $C$ is relatively large.
- Daly doesn’t ignore recovery (even though in the final formula $R$ isn’t there!)
- Daly estimates better the fraction of wasted work once a failure occurs ($T_w$)
- Daly allows failures during recovery, but not during checkpointing.
The assumption that $C << 1/\lambda = MTBF$ is likely invalid at large scale

$MTBF_{platform} = MTBF_{host}/\#hosts$

The assumption that $C$ is so small that there are no failures during checkpointing is likely invalid at large scale.

The assumption that there cannot be a failure during a downtime (of another host) is likely invalid at large scale.

"Cascading" failures

Are we even sure that periodic is optimal???
Problem statement and definition

- We focus on $C_{max}$ minimization, or rather $\mathbb{E}[C_{max}]$ minimization
  - After all, this is the real objective, not $\mathbb{E}[T_w]$
- Let us first study the case of a *sequential* job that starts at time $t_0$
  - Sounds simple, but in fact it’s already quite complicated
- Let’s not make any assumption on the distribution for now: starting at time $t_0$, failures occur at time $t_n = t_0 + \sum_{m=1}^{n} X_m$, where the $X_m$’s are *i.i.d* random variable
- $P_{suc}(x|\tau)$: the probability that there is no failure for the next $x$ seconds knowing that the last failure was $\tau$ seconds ago
  - Given by the probability distribution
Let $w$ denote an *amount of work* that remains to be done
  - i.e., a number of seconds of computation

Let $T(w|\tau)$ be the time needed to complete $w$ units of work
given that the last failure was $\tau$ seconds ago
  - Accounting for failures

Our objective: minimize $\mathbb{E}[T(W_{total}|\tau)]$

- $W_{total}$: the total amount of work to be done
- $\tau$: the number of seconds since the last failure at $t_0$
A **checkpointing strategy** is a decision procedure as follows

- Given $w$ and $\tau$, how much work $w_1$ should we attempt?
  - The attempted amount of work is called a “chunk”

- **Attempt**: repeatedly try the chunk until success
  - Success: $w_1 + C$ seconds without failure (note the $+C$)

- Then, we ask the question again for $w - w_1$ work and an updated $\tau$, until remains 0 units of work

- The checkpointing strategy chooses a sequence of chunk sizes and the number of chunks
Recursion for $T(w|\tau)$

- $T(0|\tau) = 0$ (no work is done in 0 seconds)
- $T(w|\tau) = w_1 + C + T(w - w_1|\tau + w_1 + C)$, if there is no failure in the next $w_1 + C$ seconds
  - Everything went well, we now have $w - w_1$ work to do, and the last failure is now $w_1 + C$ seconds further in the past
- $T(w|\tau) = T_{wasted}(w_1 + C|\tau) + T(w|R)$, otherwise
  - We’ve wasted a bunch of time, we still have $w$ work to do, and the last failure happened (ended) right before the last successful recovery
    - $T_{wasted}(w_1 + C|\tau)$: computation up to a failure + downtime + a recovery during which there can be failures
- We can weigh each case in the recursion by its probability...
Computing $\mathbb{E}[T]$

- Probability that there is no failure in the next $w_1 + C$ second: $P_{suc}(w_1 + C|\tau)$

- Therefore:

$$\mathbb{E}[T(W_{total}|\tau)] = P_{suc}(w_1 + C|\tau) \times (w_1 + C + \mathbb{E}[T(W_{total} - w_1|\tau + w_1 + C)]) + (1 - P_{suc}(w_1 + C|\tau)) \times (\mathbb{E}[T_{wasted}(w_1 + C|\tau)] + E(T(W_{total}|R)))$$

- Remains to compute $T_{wasted}$
Computing $T_{wasted}$, sort of

- $T_{wasted}(w_1 + C|\tau) = T_{lost}(w_1 + C|\tau) + T_{rec}$
- $T_{lost}(x|\tau)$: amount of time before a failure knowing that a failure occurs in the next $x$ seconds and that the last failure was $\tau$ seconds ago
- $T_{rec}$: time spent to do the recovery ($D + R$ in the best case, possibly more if failures during recovery)
Computing $T_{rec}$

- We can compute $T_{rec}$ as a function of $T_{lost}$:

$$T_{rec} = \begin{cases} 
D + R & \text{with probability } P_{suc}(R\mid 0), \\
D + T_{lost}(R\mid 0) + T_{rec} & \text{with probability } 1 - P_{suc}(R, 0).
\end{cases}$$

- If there is no failure for $R$ seconds right after the downtime (probability $P_{suc}(R\mid 0)$), then the recovery takes time $D + R$.

- If there is a failure (probability $1 - P_{suc}(R, 0)$), then we spend $D$ seconds of downtime, waste $T_{lost}(R\mid 0)$ seconds trying a recovery that would have lasted $R$ seconds if successful, and then we still have to recover anyway, which requires $T_{rec}$ seconds.
Weighing both cases by their probabilities we have

\[
\mathbb{E}[T_{\text{rec}}] = P_{\text{suc}}(R|0) \times (D + R) + \\
(1 - P_{\text{suc}}(R|0)) \times (D + \mathbb{E}[T_{\text{lost}}(R|0)] + \mathbb{E}[T_{\text{rec}}])
\]

which gives us:

\[
\mathbb{E}[T_{\text{rec}}] = D + R + \frac{1 - P_{\text{suc}}(R|0)}{P_{\text{suc}}(R|0)} (D + \mathbb{E}(T_{\text{lost}}(R|0)))
\]

and thus

\[
\mathbb{E}[T_{\text{wasted}}(w_1 + C|\tau)] = \\
\mathbb{E}[T_{\text{lost}}(w_1 + C|\tau)] + D + R + \frac{1 - P_{\text{suc}}(R|0)}{P_{\text{suc}}(R|0)} (D + \mathbb{E}(T_{\text{lost}}(R|0)))
\]
Putting it all, \( \mathbb{E}[T(W_{total}, \tau)] \)

\[
\mathbb{E}[T(W_{total} | \tau)] = \\
P_{suc}(w_1 + C | \tau) \times (w_1 + C + \mathbb{E}[T(W_{total} - w_1 | \tau + w_1 + C)]) + \\
(1 - P_{suc}(w_1 + C | \tau)) \times (\mathbb{E}[T_{lost}(w_1 + C | \tau)] + D + R + \\
\frac{1 - P_{suc}(R|0)}{P_{suc}(R|0)} (D + \mathbb{E}[T_{lost}(R|0)]) + \mathbb{E}[T(W_{total} | R)])
\]

Easy, right? 😊

- **Goal:** Find \( w_1 \) that minimizes the above expression
  - Note the recursion (yikes!)
- **Remains:** To know how to compute \( P_{suc}(x|y) \) and \( \mathbb{E}[T_{lost}(x|y)] \)
- **And so we make assumptions about the failure distribution...**
With exponentially distributed inter-failure times \( P(X = t) = \lambda e^{-\lambda t} \) we obtain:

\[
\mathbb{E}[T_{\text{lost}}(x|y)] = \int_0^\infty tP(X = t| t < x)dt = \frac{1}{\lambda} - \frac{x}{e^{\lambda x} - 1}
\]

\[
P_{\text{suc}}(x|y) = e^{-\lambda x}
\]

Both expressions above do not involve \( y \) because the Exponential distribution is memoryless.

So we can remove all the "|\( \tau \)" in all probabilities or expectations to simplify notations.
Exponential failures

\[ \mathbb{E}[T(W_{total})] = e^{-\lambda(w_1+C)}(w_1 + C + \mathbb{E}[T(W_{total} - w_1)]) + (1 - e^{-\lambda(w_1+C)})(\frac{1}{\lambda} - \frac{w_1+C}{e^{\lambda(w_1+C)}-1} + D + R + \frac{1-e^{-\lambda R}}{e^{-\lambda R}}(D + \frac{1}{\lambda} - \frac{R}{e^{\lambda R}-1} + \mathbb{E}[T(W_{total})]))) \]

- Assume that there are \( K \) chunks
- We can write an equation for \( \mathbb{E}[T(W_{total})] \) as a function of \( \mathbb{E}[T(W_{total} - w_1)] \)
- We can write an equation for \( \mathbb{E}[T(W_{total} - w_1)] \) as a function of \( \mathbb{E}[T(W_{total} - w_1 - w_2)] \)
- ...
- We can then solve the recursion!!
  - A LOT of (easy but very tedious) math
We obtain a general form for $\mathbb{E}[T(W_{total})]$:

$$\mathbb{E}[T(W_{total})] = A \times \sum_{i=1}^{K} (e^{\lambda(w_i+C)} - 1)$$

- $e^{\lambda(w_i+C)}$ is a convex function of $w_i$
- Therefore, $\mathbb{E}[T(W_{total})]$ is minimized when all $w_i$’s are equal
- After decades of periodic checkpointing research, we finally know that it’s optimal!! (for exponential failures)

We can also compute the optimal makespan (more math, with numerical evaluation of the Lambert function defined as $L(z)e^{L(z)} = z$)

Important: we made NO approximations
For parallel jobs a similar result is obtained
- Replace $\lambda$ by $p\lambda$ ($p$ is the number of processors)

Several possible models for $W_{total}(p)$, $C(p)$, $R(p)$, but the general spirit of the result is unchanged

One difference: we can no longer compute the optimal makespan
- We have "cascading failures": a processor fails while another one is already experiencing a downtime
- Computing $E[T_{rec}]$ becomes intractable
What is the impact of this result?

If you’re a theoretician, this is a really nice result
  - Optimality results are always nice
  - One that’s "assumed but not proven" for decades is nicer

If you’re a practitioner you can say: "who cares? we did periodic already"

If you’re a practitioner you can say: "who cares? failures are not exponentially distributed anyway"

The life of the theoretician is not easy 😊
Although Exponential failures are interesting in theory, they do not occur in practice
- Memorylessness simply doesn’t hold

A distribution that’s been proposed instead is **Weibull**

\[ P(x) = \frac{1}{\lambda} \left( \frac{x}{\lambda} \right)^{k-1}, \text{ } k \text{ is the } \textit{shape parameter} \]

- \( k = 1 \): Exponential distribution
- \( k < 1 \): The longer since the last failure, the longer until the next failure, i.e., aging well (like many computers)
- \( k > 1 \): The longer since the last failure, the sooner the next failure, i.e., aging poorly (like most cars)
Several authors have studied failure logs from actual parallel platforms. They have found reasonable good fits to Weibull distributions:

- Heath et al., SIGMETRICS 2002: $k = 0.7$, $k = 0.78$
- Liu et al., IPDPS 2008: $k = 0.51$
- Schroeder et al., DSN 2006: $k = 0.33$, $k = 0.49$

In all these works $k < 0.8$

Therefore, real-world failures are far from being Exponential, and thus far from being memoryless.
Unfortunately, with Weibull failures things are not as "easy" as for Exponential failures.

There is no reason for the optimal to use a single chunk size (i.e., to be periodic).

Due to non-memorylessness, a chunk size should be computed based on what has happened in the past.

The recursion leads to:

\[
\mathbb{E}[T^{opt}(W_{total} | \tau)] = \min_{0 < w_1 \leq W_{total}} \left[ P_{suc}(w_1 + C | \tau) \times (w_1 + C + \mathbb{E}[T^{opt}(W_{total} - w_1 | \tau + w_1 + C)]) + (1 - P_{suc}(w_1 + C | \tau)) \times (\mathbb{E}[T_{lost}(w_1 + C | \tau)] + \mathbb{E}[T_{rec}] + \mathbb{E}[T^{opt}(W_{total} | R)]) \right]
\]
The previous (recursive) problem can be solved via dynamic programming.

Given a time quantum $u$ (i.e., $w_i = n \times u$), one can design a dynamic programming algorithm with complexity $O\left(\frac{W_{total}^3}{u} \left(1 + \frac{C}{u}\right)\right)$.

This is quite expensive if $u$ is small, and an approximation. But better than saying "we don't know". As we'll see, only very few authors propose something for Weibull failures.
A simpler problem

Another problem: maximize $W$, the amount of work done before the next failure occurs

- Seems like a reasonable heuristic

Less complex recursion (sequential case shown below):

\[ \mathbb{E}[W(w|\tau)] = P_{suc}(w_1 + C|\tau)(w_1 + \mathbb{E}[W(w-w_1|\tau+w_1 + C)] \]

- Sequential case: dynamic programming in $O\left(\frac{W^3}{u}\right)$

- Parallel case: dynamic programming in $O\left(p\frac{W^3}{u}\right)$

- In practice, with several optimizations, these algorithms can run in a few seconds for realistic problem instances
Some simulation results

Evaluated checkpointing strategies

- **YOUNG**: Periodic $\sqrt{2 \times \frac{C(p)}{(\lambda p)}}$
- **DALYLOW**: Periodic $\sqrt{2 \times C(p) \times \left(\frac{1}{\lambda p} + D + R(p)\right)}$
- **DALYHIGH**: Periodic $\sqrt{2C(p)/\lambda \left[1 + \sqrt{\frac{C(p)\lambda}{18}} + \frac{C(p)\lambda}{18}\right]} - C(p)$
- **DPNEXTFAILURES**: dyn. prog. to maximize $W$
  - **DPMMAKESPAN** too expensive
- **BESTPER**: exhaustive search for the best period
  - To see how good periodic can be
Some simulation results

Evaluated checkpointing strategies

- **BOUGUERRA:**
  - A periodic policy in Bouguerra et al. [PPAM 2010]
  - Rejuvenates all processors after every failures, which makes no sense if $k < 1$

- **Liu:**
  - A non-periodic policy by Liu et al. [IPDPS 2008]
  - Specifically designed for Weibull failures
  - Leads to poor results in our experiments (error?)
  - no results presented here
Some simulation results

Platform models

- **Petascale:**
  - Inspired by the Jaguar platform: up to \( p_{max} = 45,208 \) processors
  - \( W_{total}/p_{max} \) is around 8 days
  - \( D = 60, \ C = R = 600s, \ \lambda = 125y, 500y \)

- **Exascale:**
  - Up to \( p_{max} = 2^{20} \) processors
  - \( W_{total}/p_{max} \) is around 3.5 days
  - \( D = 60, \ C = R = 600s, \ \lambda = 1250y \)
Weibull failures

- Perfectly i.i.d. Weibull, with $k = 0.7$
- Perfectly i.i.d. Weibull, with $0.1 \leq k \leq 1$

Log-based failures

- Clusters #18 and #19 from *The failure trace archive*
- Clusters with > 1000 nodes
- Used to determine empirical $P_{suc}$ values
Some simulation results

Performance metric

- For each set of parameters, we generate 600 random scenarios (i.e., 600 sets of failure traces)
- For each scenario we simulate application execution for each checkpointing strategy
- For each scenario we compute each strategy’s degradation

  - The ratio of the strategy’s makespan to that of the best strategy for that scenario
  - 1 is the best possible value,
  - 1.15 means "15% slower than the best strategy"
  - We report on the average degradation, averaged over all 600 scenarios
Some simulation results

Weibull $k = 0.7$, petascale

![Graph showing average makespan degradation against number of processors for different scenarios: BESTPer, YOUNG, DALYLow, DALYHigh, BOUGUERRA, and DPNEXTFailure.](image)
Some simulation results

Weibull $k = 0.7$, exascale

![Graph showing simulation results for different number of processors and average makespan degradation. The graph compares various algorithms including BESTPer, YOUNG, DALYLOW, DALYHIGH, BOUGUERRA, and DPNEXTFAILRE. The x-axis represents the number of processors, ranging from $2^{14}$ to $2^{20}$, and the y-axis represents the average makespan degradation, ranging from 0.9 to 1.1. The graph illustrates how the average makespan degradation increases as the number of processors increases.]
Some simulation results

Weibull $0.1 \leq k \leq 1$, petascale

![Graph showing average makespan degradation vs. Weibull shape parameter (k)]
Some simulation results

Real log, petascale

The diagram shows the average makespan degradation as a function of the number of processors.

- **BESTPER**
- **YOUNG**
- **DALYLLOW**
- **DALYHIGH**
- **DPNEWXFAILU**

Graph axes:
- Y-axis: Average makespan degradation
- X-axis: Number of processors

The graph illustrates the performance of different scenarios under varying processor numbers.
Conclusion

Important results

- Periodic checkpointing is optimal for memoryless failures.
- For non-memoryless failures, instead of minimizing makespan one can maximize the amount of work done before the next failure.
  - This is feasible via dynamic programming.
- For non-memoryless failures, in ideal Weibull scenarios, the best periodic strategy is close to the optimal.
- In real-world cases, periodic checkpointing can be far from optimal.
Sources and acknowledgments

- Checkpointing strategies for parallel jobs [SC 2011]
  M. Bougeret, H. Casanova, M. Rabie, Y. Robert, F. Vivien