Resilient application co-scheduling with processor redistribution

Anne Benoit[1]
Loïc Pottier[1]   Yves Robert[1,2]

[1]ENS Lyon & INRIA, France

[2]University of Tennessee Knoxville, USA

Anne.Benoit@ens-lyon.fr

October 4, 2016 – CCDSC 2016

Why co-scheduling?

- Supercomputers use more and more accelerators
- For instance, next supercomputer hosted by Argonne:
  - Aurora $\rightarrow$ 180 petaflops **only** provided by Xeon Phi
- One KNL (actual Xeon Phi) has 288 threads

Why co-scheduling?

- Supercomputers use more and more accelerators
- For instance, next supercomputer hosted by Argonne:
  - Aurora → 180 petaflops **only** provided by Xeon Phi
- One KNL (actual Xeon Phi) has 288 threads

More and more concurrency available ☺

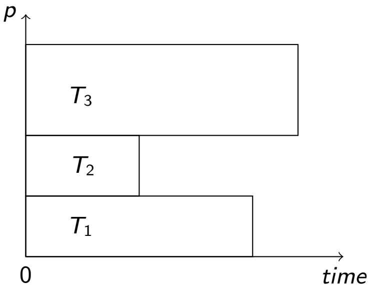
Why co-scheduling?

- Supercomputers use more and more accelerators
- For instance, next supercomputer hosted by Argonne:
  - Aurora $\rightarrow$ 180 petaflops **only** provided by Xeon Phi
- One KNL (actual Xeon Phi) has 288 threads

More and more concurrency available ☺

We want to execute applications concurrently!

Why co-scheduling?



Why resilience?

- Supercomputers enroll huge number of processors
- More components $\rightarrow$ increased probability of errors
- MTBF of 1 processor $\rightarrow$ around 100 years
- MTBF of p processors $\rightarrow$ $\frac{100}{p}$
- MTBF Titan $<$ 1 day

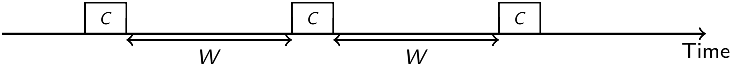
Why resilience?

- Supercomputers enroll huge number of processors
- More components $\rightarrow$ increased probability of errors
- MTBF of 1 processor $\rightarrow$ around 100 years
- MTBF of p processors $\rightarrow \frac{100}{p}$
- MTBF Titan < 1 day

Resilience at petascale is **already** a problem ☹

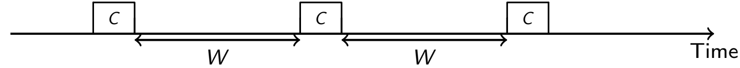
Checkpoint with fail-stop errors

Save the state of the application periodically:

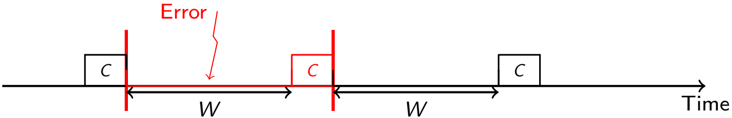


Checkpoint with fail-stop errors

Save the state of the application periodically:

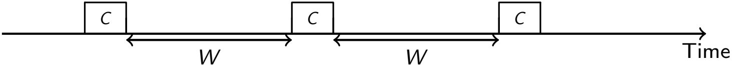


In case of errors, application returns to last checkpoint:

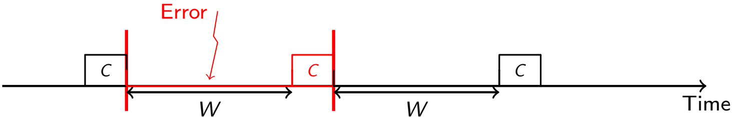


Checkpoint with fail-stop errors

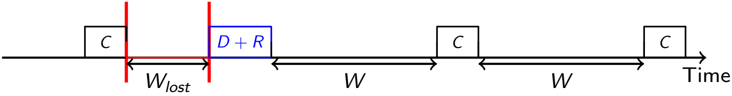
Save the state of the application periodically:



In case of errors, application returns to last checkpoint:



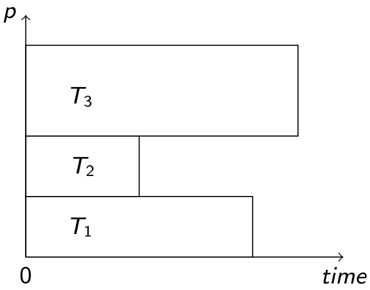
Work done between last checkpoint and error is lost;
downtime D and *recovery R* before resuming execution:



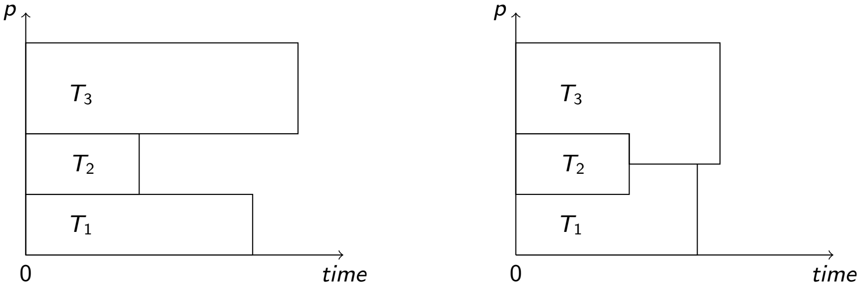
Outline

- Model and complexity
- Heuristics
- Simulation results
- Conclusion

Example

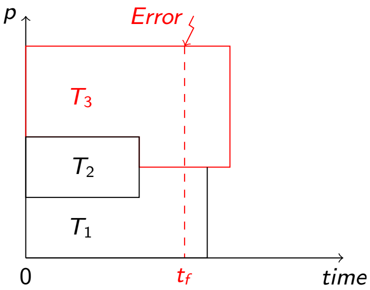


Example

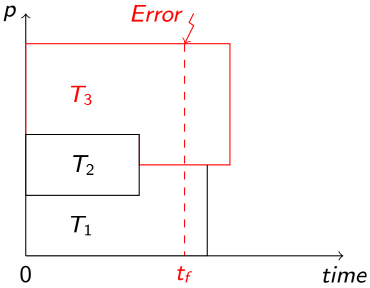


Redistribution when T_2 releases its processors

Example



Example



How to compute the new execution time of T_3?

Give processors of T_1 to T_3?

Model

- n independent parallel applications $T_1, T_2, \ldots, T_n$
- Execution platform with p identical processors
- Each application is *malleable*: its number of processors j can change at any time
- Each application is a divisible load application

Problem: CoSched

Minimize the maximum of the expected completion times of n applications executed on p processors subject to failures. Redistributions are allowed only when an application completes execution or is struck by a failure.

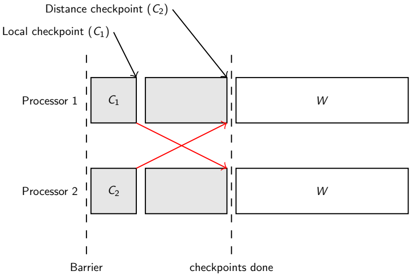
Fault model

- Only *fail-stop* errors
- Errors follow an exponential law $Exp(\lambda)$
  - Mean Time Between Faults (MTBF) for one proc.: $\mu = 1/\lambda$
  - For application T_i with j processors: $\mu_{i,j} = \mu/j$
- Use of light-weight periodic checkpointing protocol, with period $\tau_{i,j} = \sqrt{2\mu_{i,j}C_{i,j}} + C_{i,j}$ [Young, 1974], where $C_{i,j}$ is the checkpoint cost
- $C_{i,j} = \frac{m_i}{j\tau} + \beta$, where m_i is the memory footprint of T_i, β is a start-up latency and τ is the link bandwidth

Checkpointing model

Double checkpointing algorithm [Kalé et al. 2004]

- Each processor stores two checkpoints: its own and that of its *buddy* processor
- If there is a fault, the buddy processor sends back both checkpoints



The number of processors allocated to each application is **even**

Execution time

For application T_i with j processors:

- Fault-free execution time: $t_{i,j}$
- Resilient expected execution time: $t^R_{i,j}(\alpha_i)$, where α_i is the remaining fraction of work that needs to be executed by T_i (initially, $\alpha_i = 1$)
- We can easily express the number of checkpoints, $N^{\text{ff}}_{i,j}(\alpha_i)$, and then obtain an expression of $t^R_{i,j}(\alpha_i)$:

$$t^R_{i,j}(\alpha_i) = e^{\lambda j R_{i,j}}\left(\frac{1}{\lambda j} + D\right)\left(N^{\text{ff}}_{i,j}(\alpha_i)(e^{\lambda j \tau_{i,j}} - 1) + (e^{\lambda j \tau_{last}} - 1)\right)$$

With redistribution

- Redistribution done (i) when an application ends, or (ii) when an error strikes

- Redistribution cost of application T_i from j to k processors $RC_i^{j \to k}$ depends on:
  - Data footprint of T_i (m_i)
  - Number of processors involved (j to k)
  - Link bandwidth τ, start-up latency β
  - Constant start-up overhead S

$$RC_i^{j \to k} = S + \max(\min(j, k), |k - j|) \times \left(\frac{m_i}{kj\tau} + \beta \right)$$

  - After redistribution, we systematically checkpoint and therefore pay the cost $C_{i,k}$

Remaining fraction of work at time t

- Initially, $\alpha_i = 1$ for $1 \leq i \leq n$, and we remove progressively the work already completed
- Time when last redistribution or failure occurred for application T_i: t_{lastR_i}
- Number of checkpoints between t_{lastR_i} and the event at time t: $N_{i,j} = \left\lfloor \frac{t - t_{lastR_i}}{\tau_{i,j}} \right\rfloor$

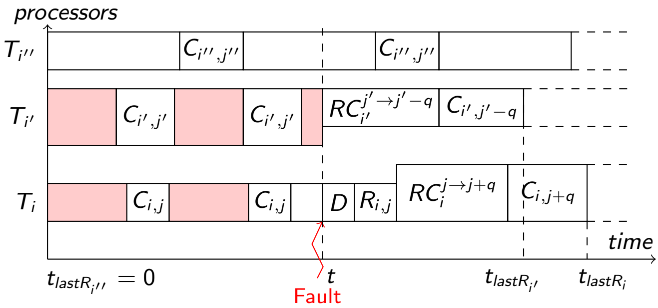
Remaining fraction of work at time t

- Initially, $\alpha_i = 1$ for $1 \leq i \leq n$, and we remove progressively the work already completed
- Time when last redistribution or failure occurred for application T_i: t_{lastR_i}
- Number of checkpoints between t_{lastR_i} and the event at time t: $N_{i,j} = \left\lfloor \frac{t - t_{lastR_i}}{\tau_{i,j}} \right\rfloor$

How to compute the α_i values,
and hence the expected execution times of applications?

Computation of work done

Example of redistribution when a fault strikes application T_i: the colored rectangles correspond to useful work done by T_i and $T_{i'}$ before the failure; $T_{i''}$ is not affected by the failure (no redistribution)



- If T_i is the faulty application: $\alpha_i = \dfrac{N_{i,j} \times (\tau_{i,j} - C_{i,j})}{t_{i,j}}$
- Otherwise: $\alpha_i = \dfrac{t_f - t_{lastR_i} - N_{i,j} C_{i,j}}{t_{i,j}}$

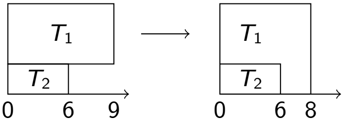
Complexity without redistribution

Theorem 1

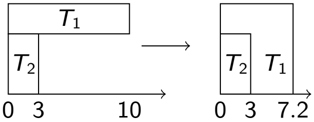
The CoSched problem without redistributions can be solved in polynomial time $O(p \times \log(n))$, where p is the number of processors, and n is the number of applications

- Each application has two processors
- We allocate the $p - 2n$ remaining processors two by two in a greedy way to longest application

$$T_1 = \begin{cases} t_{1,1} = 10, & w_{1,1} = 10 \\ t_{1,2} = 9, & w_{1,2} = 18 \\ t_{1,3} = 6, & w_{1,3} = 18 \end{cases} \qquad T_2 = \begin{cases} t_{2,1} = 6, & w_{2,1} = 6 \\ t_{2,2} = 3, & w_{2,2} = 6 \\ t_{2,3} = 3, & w_{2,3} = 9 \end{cases}$$



(a) Greedy uses largest execution time to allocate processors



(b) Greedy-SP uses best speedup profile to allocate processors

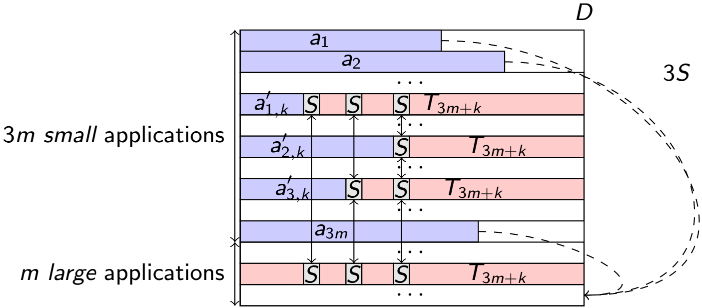
Some examples where Greedy-SP is not optimal either...

Complexity with redistribution

Theorem 2

With constant redistribution costs and without failures, CoSched is NP-complete (in the strong sense)

Reduction from 3-Partition with **distinct integers**



Algorithms and heuristics

Optimal greedy algorithm without redistribution to allocate processors to applications at beginning

Two cases of redistribution:

- When an application ends and releases its processors
- When a fault occurs, we redistribute only if the faulty application becomes the longest one

Heuristics

Two heuristics when applications end:

- EndGreedy: Greedy algorithm with redistribution costs
- EndLocal: Local decisions (take processors from shortest applications)

Two heuristics in case of fault:

- IteratedGreedy: Greedy algorithm with redistribution costs
- ShortestAppsFirst: Local decisions (take processors from shortest applications)

Test platform

- Fault simulator, synthetic applications

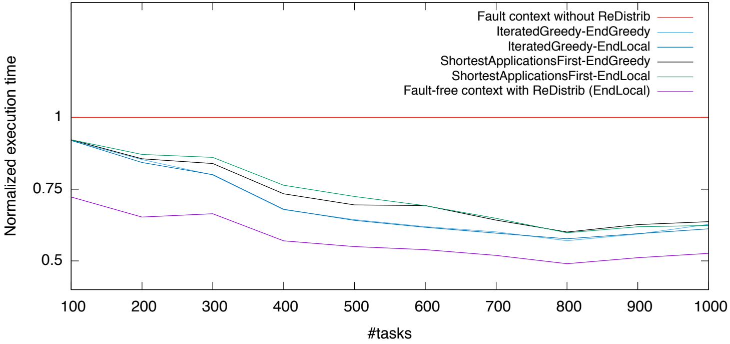
Fault-free execution time (Amdahl model)

$$t_{i,1} = 2 \times m_i \times log_2(m_i)$$

$$t_{i,j} = f \times t_{i,1} + (1-f)\frac{t_{i,1}}{j} + \frac{m_i}{j} log_2(m_i)$$

- m_i: number of data needed by application i
- f: sequential fraction of time ($f = 0.08$ for our tests)

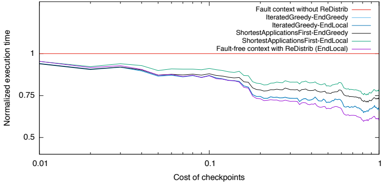
Impact of *n* with 5000 processors and an MTBF of 100 years for each processor

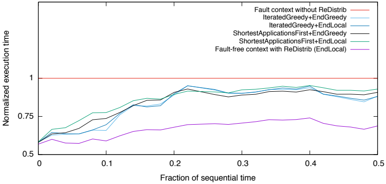


Heuristics are more efficient when the number of applications increases. With $n = 1000$, we obtain a gain around 40%.

Results

Impact of checkpointing cost c and sequential fraction f
with $n = 100$ and $p = 1000$





Heuristics more efficient when checkpointing cost decreases.
Heuristics *very* efficient when applications are almost fully parallel.

Summary of results

- IteratedGreedy better than ShortestAppsFirst: rebuilds complete schedule at each fault (except for very low MTBF, 10 years or less)
- Faulty context: gain flexibility from failures
- Too many processors/too few applications: less need of redistribution
- Best context: heterogeneous applications
- Significant impact of checkpointing cost and fraction of sequential time
- All heuristics run within a few seconds, while total execution time of applications takes several days: negligible overhead

Conclusion

- Detailed and comprehensive model for scheduling a pack of applications with failures and redistributions
- Greedy polynomial-time algorithm with failures but no redistribution
- With redistribution: NP-completeness of the problem, even with constant redistribution costs and no failures
- Polynomial-time heuristics to redistribute efficiently: significant improvement of execution time

Conclusion

- Detailed and comprehensive model for scheduling a pack of applications with failures and redistributions
- Greedy polynomial-time algorithm with failures but no redistribution
- With redistribution: NP-completeness of the problem, even with constant redistribution costs and no failures
- Polynomial-time heuristics to redistribute efficiently: significant improvement of execution time

Future work:

- How to partition applications into packs?
- Competitiveness of online redistribution algorithms?
- How to deal with silent errors?