Fault Tolerance (and a bit of Energy)

ICS632: Principles of High-Performance Computing

Henri Casanova (henric@hawaii.edu)

Fall 2015

Disclaimer

- These slides will provide a quick overview of many important questions faced by theoreticians and practitioners in HPC
- Most of this is still very much in the research and development stage
- It would take us way too long to explore all this content in depth

- We'll learn about some of the topics through research paper presentations

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
- Several challenges arise (topics of entire conferences/workshops!)
  - Resilience to failures
  - Power consumption
  - Heat management
  - Network interconnects
- Some for people who build platforms, some for people who use platforms

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
- Several challenges arise (topics of entire conferences/workshops!)
  - Resilience to failures
  - Power consumption
  - Heat management
  - Network interconnects
- Some for people who build platforms, some for people who use platforms

Outline

1. Why Fault Tolerance?
2. Checkpointing
3. A little theory
4. A little practice
5. Energy Consumption

Faults, Errors, and Failures

- A fault corresponds to an execution that does not go according to specification (hardware, software)
- A fault may cause an error, i.e., an incorrect system state
- An error may cause a failure, i.e., an unacceptable behavior (e.g., crash, wrong output)
- WARNING: terminology is all over the place in the literature

- Why do failures occur? Because redundancy is too costly
- General question: how to achieve reliability out of unreliable components?

All kinds of faults/failures

- Hardware faults:
  - Detected and corrected by hardware (ECC works)
  - Detected in hardware, but flagged as not corrected (e.g., double bit-flip)
  - Silent (bits are wrong, but you don't know!)
- Software faults:
  - Pure software faults (bugs)
  - Mis-handling of hardware faults
  - OS/firmware faults
  - Many of these are not silent, but not all
- Some failures can be handled by a "reboot", others by a "replace hardware"
  - In practice, "replacing" means "boot a spare"
  - So the two are very similar

How often do failures occur?

- Say you buy a component with Mean Time Between Failures (MTBF) T
- Now you put n components together in a platform
- The platform's MTBF is T/n
- When n is large, the MTBF will be low
- So we have a problem for (exa)scale

- Failure distributions are often assumed i.i.d. and exponential (memoryless) in the literature
- But in practice they are likely not i.i.d. and not memoryless (e.g., Weibull), which is much more difficult to work with for theoreticians

Outline

1 Why Fault Tolerance?

2 Checkpointing

3 A little theory

4 A little practice

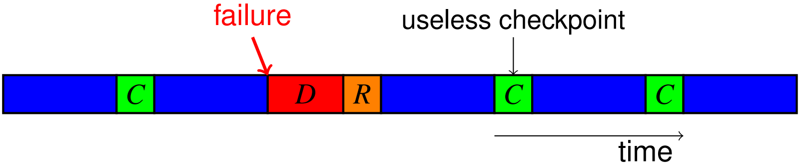
5 Energy Consumption

Checkpoint time, recovery time, down time

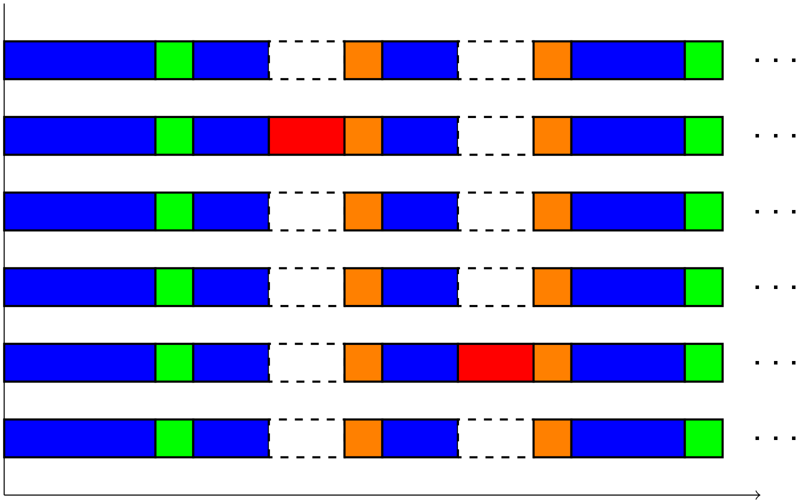
- Checkpoint time: C seconds
  - Depends on checkpoint size, storage medium
- Recovery time: R seconds
  - Depends on checkpoint size, storage medium
- Downtime: D seconds
  - After a failure, time for the *logical* host to start a recovery
  - Typically: constant time to bring a spare host on-line
-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

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
- On a single processor:



Coordinated checkpointing on multiple processors



Outline

1. Why Fault Tolerance?
2. Checkpointing
3. A little theory
4. A little practice
5. Energy Consumption

"Theoretical" questions

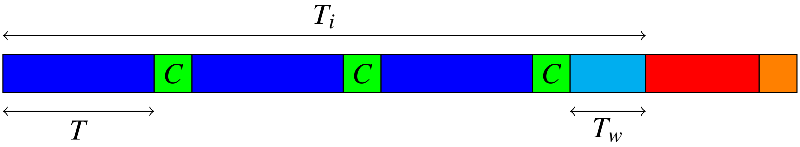
- How often should one checkpoint?
  - Goal: minimize expected makespan
- Should checkpointing be periodic?
  - What everybody has done in practice
- Let's look at some development on one processor to get a sense of what theoreticians have done

Assuming Periodic checkpointing

- Many authors have proposed analytical expressions for the "optimal" checkpointing frequency, **assuming that checkpointing must be periodic**
  - or approximations thereof
- A famous such expression is given by J.W. Young
  - *A First Order Approximation to the Optimal Checkpoint Interval*, Communication of the ACM, 1974.
- Let's review how this bound is achieved, which will provide context for recent results

Young's approach

- Let T be the time between checkpoints
- Let T_w be the compute time wasted due to a failure
- Let us consider an interval T_i in between two failures



$$T_i = n \times (T + C) + T_w$$

Young's assumptions

- Objective: minimize $\mathbb{E}[T_w]$, where $T_w = T_i - n(T + C)$
- Four questionable assumptions:
  - 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)

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: $e^{\lambda T} e^{\lambda C}(1 - \lambda T) - 1 = 0$
- λT 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 $\frac{1}{2}(\lambda T)^2 = 1 - e^{-\lambda C}$
- λC 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

Going further

- 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:

$$T_{opt} = \sqrt{2C/\lambda}\left[1 + \sqrt{\frac{C\lambda}{18}} + \frac{C\lambda}{18}\right] - C, \quad \text{if} \quad C < 2/\lambda$$

$$T_{opt} = 1/\lambda, \quad \text{if} \quad C \geq 2/\lambda$$

- 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 the fraction of wasted work once a failure occurs (T_w) much better
- Daly allows failures during recovery, **but not during checkpointing**

Research questions

- The assumption that $C << 1/\lambda = MTBF$ is likely invalid at large scale
  - $MTBF_{platform} = MTBF_{proc} / \#procs$
- 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???

Research questions

- The assumption that $C << 1/\lambda = MTBF$ is likely invalid at large scale
  - $MTBF_{platform} = MTBF_{proc}/\#procs$
- 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???

Research questions

- The assumption that $C << 1/\lambda = MTBF$ is likely invalid at large scale
  - $MTBF_{platform} = MTBF_{proc}/\#procs$
- 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???

Research questions

- The assumption that $C << 1/\lambda = MTBF$ is likely invalid at large scale
  - $MTBF_{platform} = MTBF_{proc}/\#procs$
- 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???

New 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 τ seconds ago
  - Given by the probability distribution

New 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 τ seconds ago
  - Given by the probability distribution

New 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 τ seconds ago
  - Given by the probability distribution

New 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 τ seconds ago
  - Given by the probability distribution

Problem statement and definition

- 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 τ seconds ago
  - Accounting for failures
- Our objective: minimize $\mathbb{E}[T(W_{total}|\tau)]$
  - W_{total}: the total amount of work to be done
  - τ: the number of seconds since the last failure at t_0

Problem statement and definition

- 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 τ seconds ago
  - Accounting for failures
- Our objective: minimize $\mathbb{E}[T(W_{total}|\tau)]$
  - W_{total}: the total amount of work to be done
  - τ: the number of seconds since the last failure at t_0

Problem statement and definition

- 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 τ seconds ago
  - Accounting for failures

- Our objective: minimize $\mathbb{E}[T(W_{total}|\tau)]$
  - W_{total}: the total amount of work to be done
  - τ: the number of seconds since the last failure at t_0

Checkpointing strategy

- A checkpointing strategy is a decision procedure as follows
- Given w and τ, 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 τ, 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 $\mathbb{E}[T]$

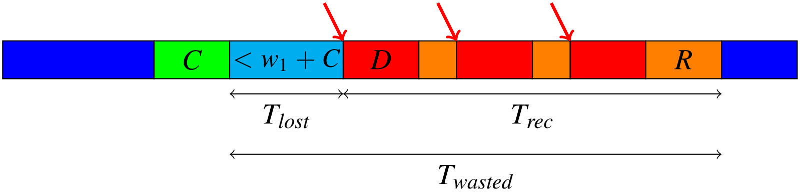
- Probability that there is no failure in the next $w_1 + C$ second: $P_{suc}(w_1 + C|\tau)$
- Therefore:

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

- 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 τ 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}:

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

- If there is no failure for R seconds right after the downtime (probability $P_{suc}(R|0)$), 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 $T_{lost}(R|0)$ 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

Computing T_{rec}

- Weighing both cases by their probabilities we have

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

  which gives us:

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

  and thus

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

Putting it all together, $\mathbb{E}[T(W_{total}, \tau)]$

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

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...

Exponential failures

- With exponentially distributed inter-failure times ($P(X = t) = \lambda e^{-\lambda t}$) we obtain:

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

- 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

- With exponentially distributed inter-failure times ($P(X = t) = \lambda e^{-\lambda t}$) we obtain:

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

- 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

Solved recursion

- 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)
- Important: we made NO approximations
- *Checkpointing Strategies for Parallel Jobs*, Bougeret, Casanova, Rabie, Robert, Vivien [Supercomputing 2011]

Solved recursion

- 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)
- Important: we made NO approximations
- *Checkpointing Strategies for Parallel Jobs*, Bougeret, Casanova, Rabie, Robert, Vivien [Supercomputing 2011]

Solved recursion

- 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)
- Important: we made NO approximations
- *Checkpointing Strategies for Parallel Jobs*, Bougeret, Casanova, Rabie, Robert, Vivien [Supercomputing 2011]

Other theoretical results, and impact

- The same result holds for parallel jobs
  - But we cannot compute the optimal makespan
- If failures are non-memoryless, then periodic is no necessarily optimal
  - The optimal checkpointing strategy can be computed via complicated dynamic programming for Weibull failures
- Nice theory, but if you're a practitioner:
  - You are doing periodic anyway because it's easy
    - Empirical results show that periodic is really bad only in corner cases
  - You know that failures are not i.i.d. anyway, so he theory doesn't apply
  - And if it did, it would be too complex (and slow to compute!)

Other theoretical results, and impact

- The same result holds for parallel jobs
  - But we cannot compute the optimal makespan
- If failures are non-memoryless, then periodic is no necessarily optimal
  - The optimal checkpointing strategy can be computed via complicated dynamic programming for Weibull failures
- Nice theory, but if you're a practitioner:
  - You are doing periodic anyway because it's easy
    - Empirical results show that periodic is really bad only in corner cases
  - You know that failures are not i.i.d. anyway, so he theory doesn't apply
  - And if it did, it would be too complex (and slow to compute!)

Outline

1. Why Fault Tolerance?
2. Checkpointing
3. A little theory
4. A little practice
5. Energy Consumption

Checkpointing overhead

- Assume you have some magical, fast way to determine the optimal checkpointing policy for realistic failures
- That's all well and good, but you cannot scale due to C being large!
  - Parallel efficiency drops as p increases (Amdahl's law)
- At large scale, your application will spend more time saving state than computing state!!!

- Crucial research problem: reduce checkpointing overhead
- Let's see what people have done/proposed

Reducing the checkpoint size

- Brute-force system-level checkpointing: save the whole address space
  - What Sean is trying to install on our Cray cluster for sequential jobs
- Problem: only a (small) fraction of the address space is needed for a recovery
  - Example: only the iteration number and 2 arrays
- More scalable application-level checkpointing: save only what data is necessary for recovery
- Drawback: you must modify application code to checkpoint
  - Note that automatic system-level checkpointing of MPI applications is tough anyway

Zero checkpoint size?

- In some (lucky?) cases, the algorithm in the application can provide clever ways to “reconstruct” a checkpoint without ever saving it!
- Algorithm Based Fault Tolerance (ABFT):
  - A technique by which the algorithm is modified, and made less efficient, to compute on encoded (with checksums) data, so that (otherwise silent) errors can be corrected
- Has been extended to deal with "fail-stop" errors
  - Maintain checksums
  - Use checksums to reconstruct lost data
- Mostly used for linear algebra applications
- Sort of like RAID 2 (parity bits, no duplication) in memory

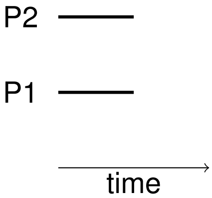
Reducing the checkpointing time

- One big problem is that when a checkpoint occurs, all processes say "save my state" at the same time!
  - e.g., if you have a NAS, the bandwidth to it becomes a massive bottleneck
- Several solutions have been proposed:
  - Use a cluster with fast storage (SAN)
  - Save only to local disk
    - But then you can't recover from a fatal hardware failure
  - Save to a "buddy" (save my state on my neighbor's disk)
  - Go diskless: keep my checkpoint in my buddy's memory
    - But then it must fit

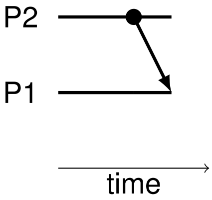
Go uncoordinated

- In many cases the "we all save at the same time" is a showstopper
- Radical idea: allow uncoordinated checkpointing:
  - Processes can checkpoint their data whenever they want/choose
  - Should create a "smooth/low" load on the storage system
- Big challenge: how do we perform recovery?
- Let's see the canonical example

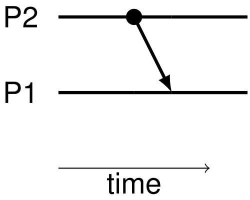
Uncoordinated checkpointing example



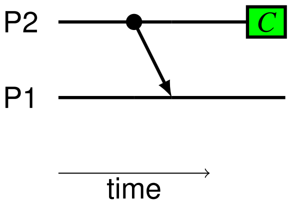
Uncoordinated checkpointing example



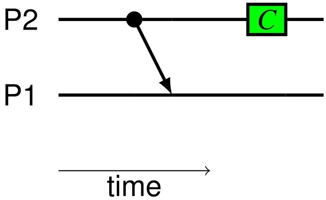
Uncoordinated checkpointing example



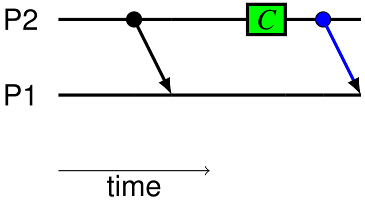
Uncoordinated checkpointing example



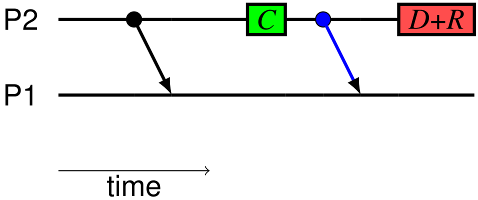
Uncoordinated checkpointing example



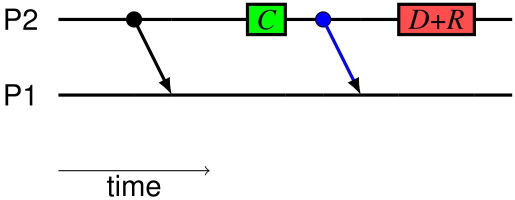
Uncoordinated checkpointing example



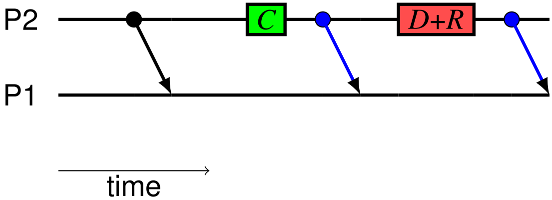
Uncoordinated checkpointing example



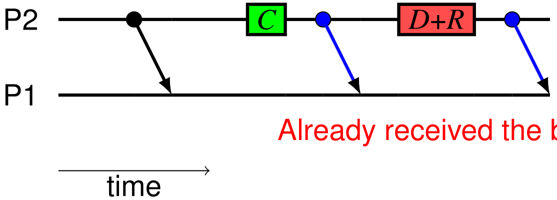
Uncoordinated checkpointing example



Uncoordinated checkpointing example



Uncoordinated checkpointing example



Already received the blue message!!

Message Logging

- To deal with the previous, and many more complex situations, we need to do some kind of message logging
- We keep track of messages sent/received, and ignore redundant copies
- How to do this efficiency and correctly is not easy
- Many "fault-tolerance MPI" implementations do this now as a matter of course

- An enormous theoretical/applied literature on this problem

Hierarchical Checkpointing: Do it all!

- The term “hierarchical" scheduling refers loosely to a solution that applies many (all?) of the previous techniques together
- And still, projections show that this won’t be enough to achieve parallel efficiency on upcoming "exascale" machines
- So we need other techniques in addition to checkpoint-rollback-recovery...

Replication

- Run redundant processes to increase the MTBF!
  - Some MPI implementations can do this transparently!
- **Good:** we can checkpoint much less frequently
- **Bad:** we "waste" resources (and power!)
- It may seem surprising that we can gain anything by wasting resources
- But it all works out because using $p/2$ processors has higher parallel efficiency than using p processors

- Let’s see an example...

Redundant processors can help!

- Processors fail exponentially with MTBF $1/\lambda$
- We have an application that runs in time $\alpha + \beta/p$ on p processors when no failures occur
- We checkpoint every $X = \sqrt{2C/\lambda}$ (Young)
- Our makespan is: $\alpha + \beta/p + \frac{\alpha+\beta/p}{X} \times C$
- Now we replicate each process, thus running on $p/2$ “more reliable processors”
- The MTBF of a pair of processors is now $(3/2)(1/\lambda)$
  - Compute integrals to verify this...
- We checkpoint every $Y = \sqrt{2C(3/2)/\lambda} = \sqrt{3C/\lambda}$
- Our makespan is now: $\alpha + 2\beta/p + \frac{\alpha+2\beta/p}{Y} \times C$

Redundant processors can help! (2)

- The math on the previous slide is really back-of-the-envelope (e.g., makespan values are for the failure-free case)
- Let's pick values : $\alpha = 1$, $\beta = 100$, $C = 10$, $\lambda = 0.001$
- For $p = 1000$
  - No replication: 1.125 time units
  - Replication: 1.222 time units (replication hurts)
- For $p = 100000$
  - No replication: 1.022 time units (horrendous efficiency)
  - Replication: 1.018 time units (replication helps)

Failure Prediction

- Another idea to use in conjunction with checkpointing is Failure Prediction
- Hardware sensors can provide a good sense of when a processor might fail in the future
  - Based on “precursor” events
- Simple idea: checkpoint proactively only when a failure seems likely in the short term
- Raises all types of interesting questions about false positive and false negative

- This is often called “failure avoidance”

What about silent errors?

- Dealing with silent errors has become a hot research topic
- Simple idea: perform (periodic?) checks on the data
- Trade-off:
  - Checking data infrequently leaves you open to a lot of waste
  - Checking data frequently has a lot of overhead
- Must be combined with checkpointing
- Typically we have various "checkers", some expensive and accurate, some cheap and less accurate
- Problem: deciding when and how to check for silent errors

Fault-Tolerance for parallel applications

- This was a whirlwind tour of fault-tolerance
- It is a huge area of research and development, that goes from pure theory to pure engineering
- Hardware developments shape the field constantly
- There is no current consensus on what will work for exascale, but there are many pathways
- Every month new papers are being published :)

- One recent workshop report: *Addressing failures in exascale computing*, Snir et al., IJHPCA 28(2), 129–173, 2014.

Outline

1 Why Fault Tolerance?

2 Checkpointing

3 A little theory

4 A little practice

5 Energy Consumption

Power is money

- Power budget is a key issue for large-scale platform
  - Hence building them next to powerplants
- Many engineering issues to reduce power consumption:
  - Microprocessor design
  - Novel cooling approaches
  - Green energies
  - ...
- But given a system with some power-management “knobs," what do you do for a given application?
  - e.g., is your fault tolerant solution too power-hungry?
  - e.g., does your idea to decrease makespan by 1% increase power consumption by 50%?

DVFS

- Common power-management techniques are Dynamic Voltage Scaling (DVS) and Dynamic Frequency Scaling
- Modern microprocessors allow voltage/frequency to be modified in software
- Without getting into details: you can slow down your computation and save on energy
  - e.g., Power consumption is a polynomial function of the frequency / voltage
- The question then becomes: by how much should I slow down my nodes to achieve energy and performance goals?
- Interestingly, if I take the voltage too low, then I can create silent errors (i.e., computation is wrong)

A slew of (interesting) problems

- Say you have a parallel application and you:
  - Can run on some number of processors in a platform with DVFS-enabled processors
  - Use process duplication or not
  - Use some complex hierarchical checkpointing
  - Want to detect silent errors and have a bunch of detectors of various cost and accuracy
- How to decide on all the above?
- Sample research questions:
  - Q: minimize makespan for a given energy budget
  - Q: minimize energy given a makespan bound
  - Q: mimimize chances of having a wrong result given a makespan bound and budget

A very active research area

- Questions pertaining to fault tolerance and to energy are being investigated extremely actively
- We only scratched the surface here
- But some of the papers you'll present this semester touch on concerns of fault tolerance and power...