Lecture 9:
Workload-Driven Evaluation

MMMMMM.... Valentine's Day coding

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Announcements

- Assignment 1 graded, solutions available (here)
  - mean 53/60, median 55/60

- GTX 480 GPUs installed in GHC 5205

15-418 road map

Modern multi-core chip architectures: multi-core + SIMD + threads

Ways to think about parallelism and communication

- At the architectural level -- machine styles: shared memory, message passing
- At the abstraction level -- programming models: shared memory, message passes, data parallelism

How to write and optimize parallel programs

⋮

An aside on GPU programming

Case studies and example techniques

Today → Evaluating system performance

Shared address space implementation details

⋮

Exam I (March 6)

You are hired by [insert your favorite chip company here].

You walk in on day one, and your boss says
"All of our senior architects have decided to take the year off.
Your job is to lead the design of our next parallel processor."

What questions might you ask?

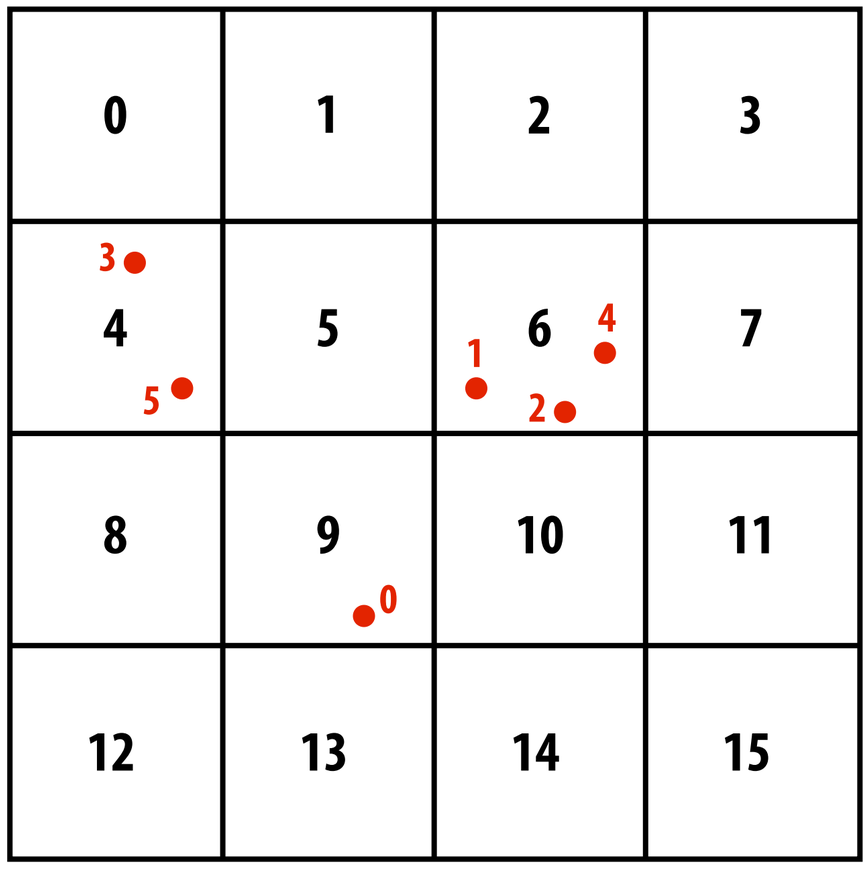
Your boss selects the program that matters most to the company
"I want you to demonstrate good performance on this application."

- **Absolute performance**
  - Often measured as wall clock time
  - Another example: operations per second

- **Speedup: performance improvement due to parallelism**
  - Execution time of sequential program / execution time on P processors
  - Operations per second on P processors / operations per second of sequential program

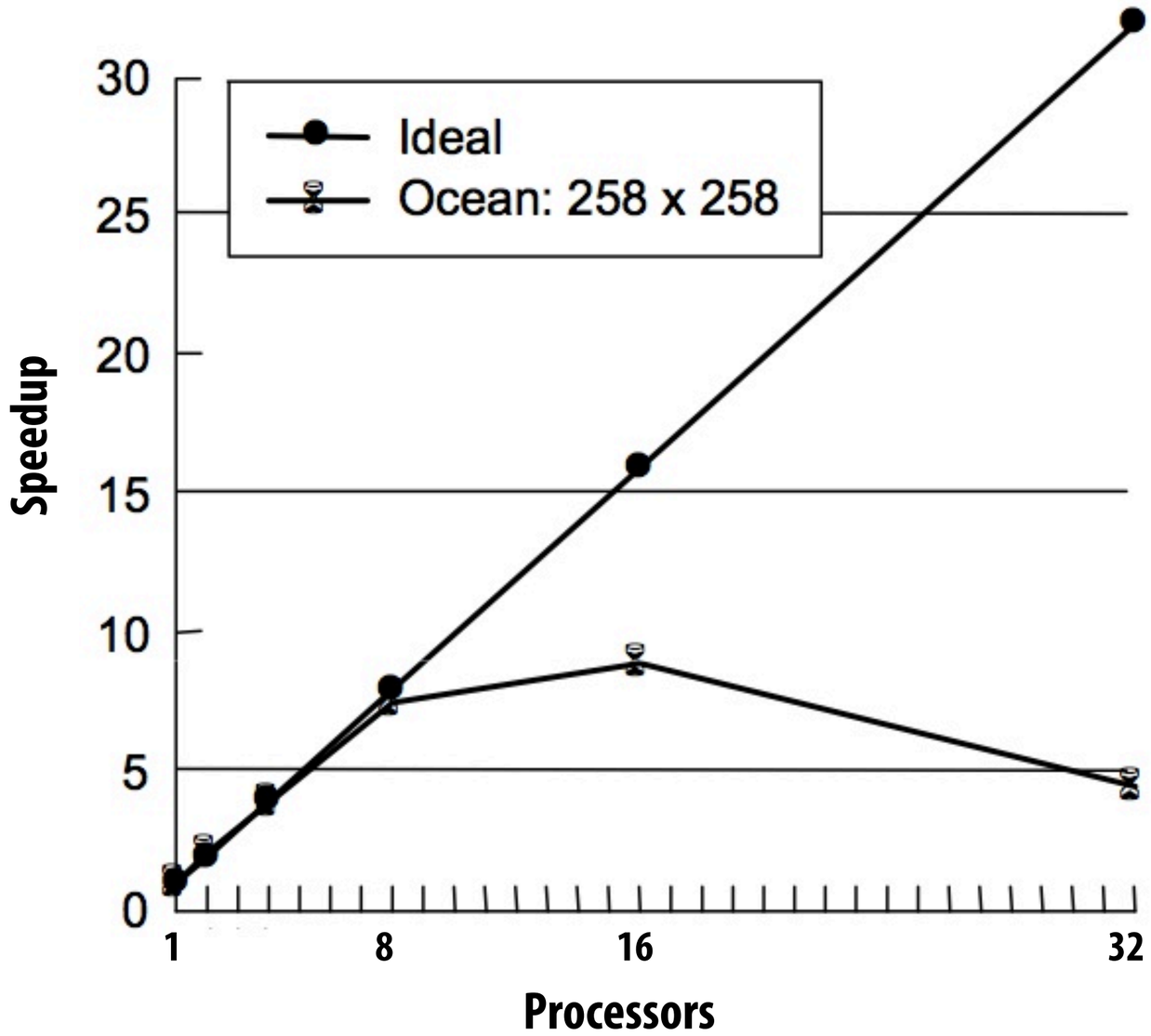
Scaling

- Should speedup be measured against the parallel program running on one processor, or the best sequential program?
  - Example: particle binning problem from last time (data-parallel algorithm would not be used on a single processor system)



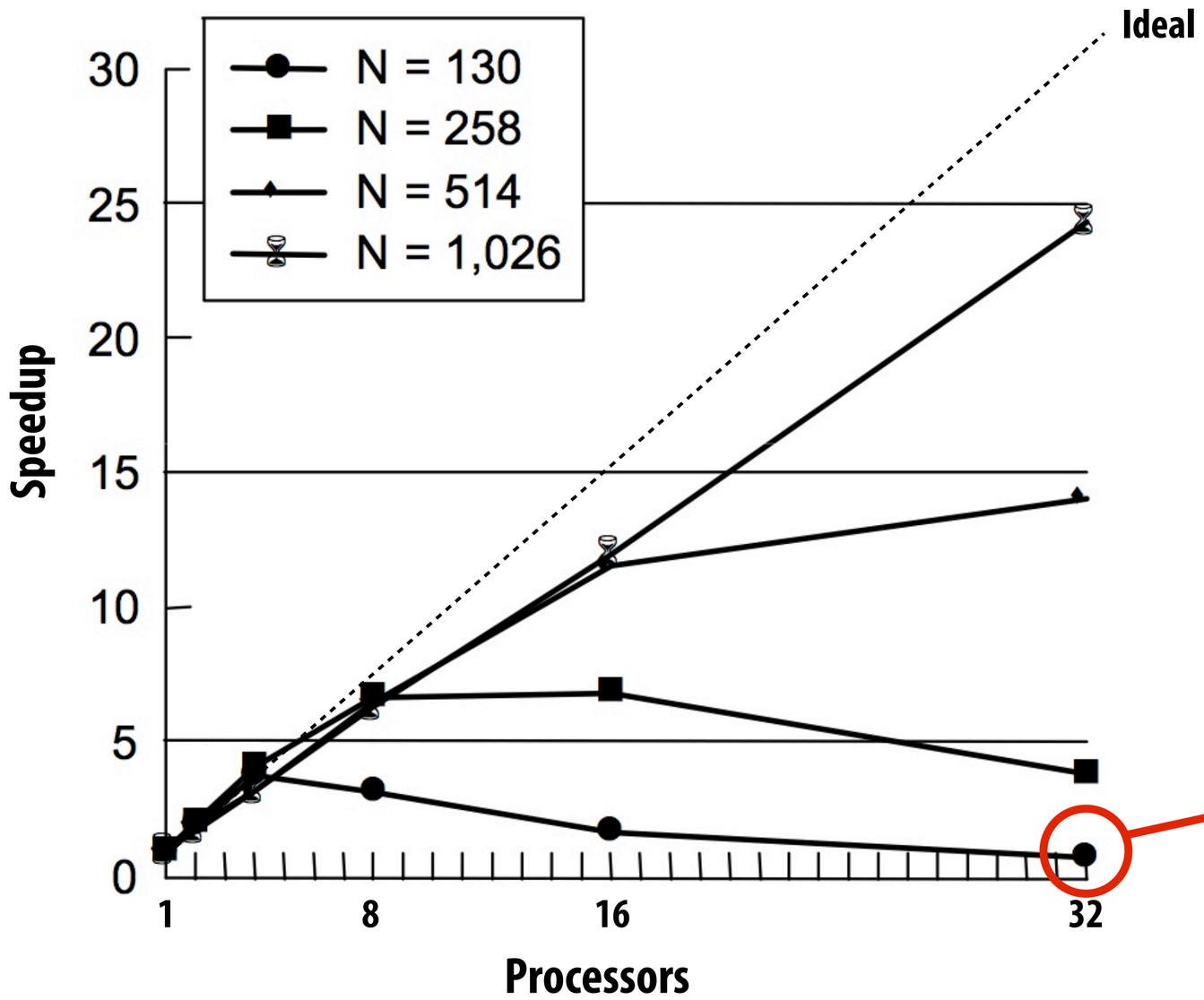
Speedup of ocean application: 258 x 258 grid

Execution on 32 processor SGI Origin 2000



Pitfalls of fixed problem size speedup analysis

Execution on 32 processor SGI Origin 2000



no benefit (slight slowdown)

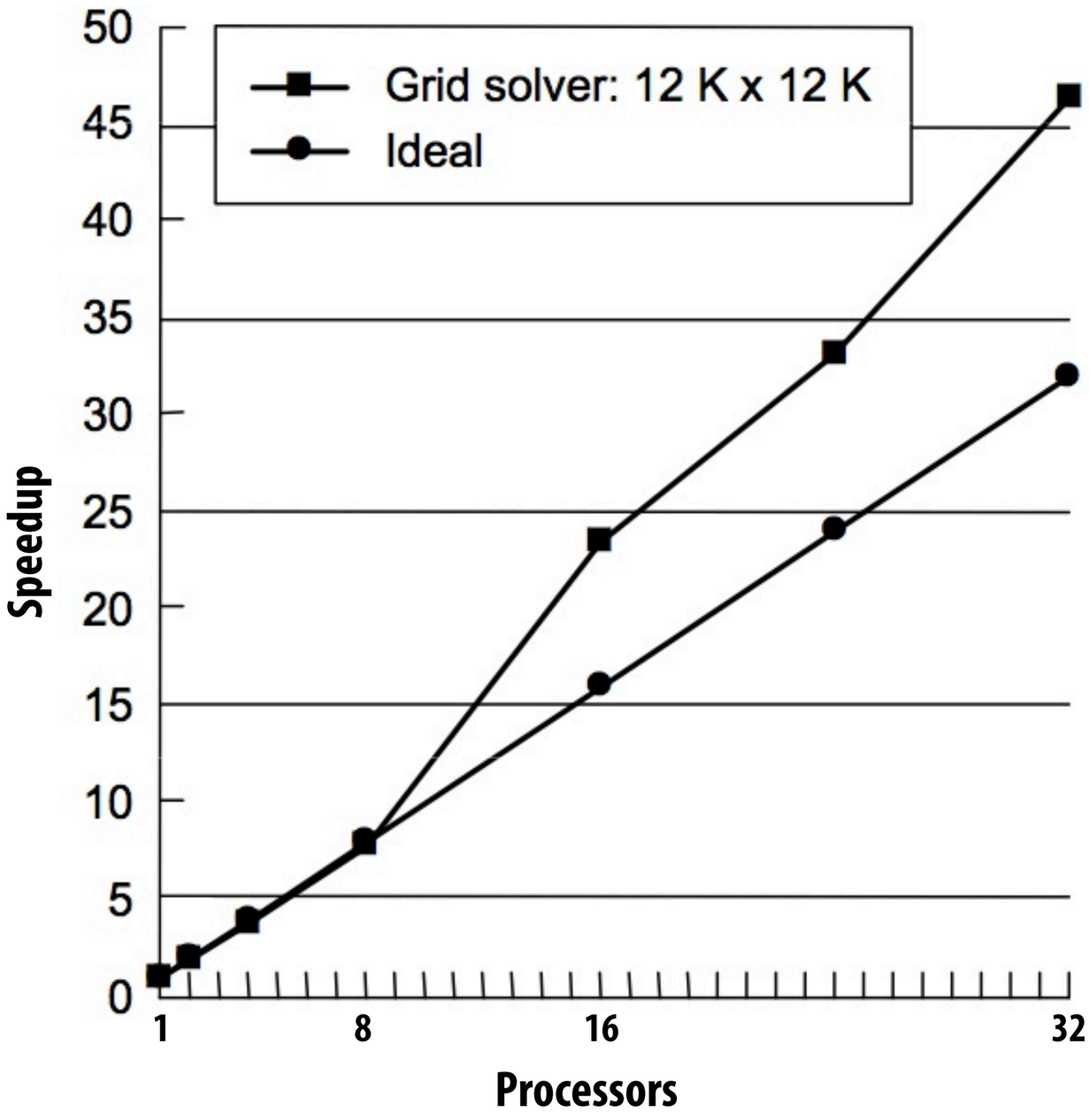
Problem is just too small for the machine

258 x 258 grid on 32 processors: ~ 310 grid cells per processor

1K x 1K grid on 32 processors: ~ 32K grid cells per processor

Pitfalls of fixed problem size speedup analysis

Execution on 32 processor SGI Origin 2000



Super-linear speedup: with enough processors, key working set fits in cache

Another example: if problem size is too large, working set may not fit in sequential computer's memory: paging to disk

(this would make speedup look amazing)

Understanding scaling: size matters

- Application and system scaling parameters have complex interactions
  - Impact load balance, overhead, communication-to-computation ratios (inherent and artifactual), locality of data access
  - Effects often dramatic, sometimes small, application dependent

- Evaluating a machine with a fixed problem size can be problematic
  - Too small a problem:
    - Parallelism overheads dominate parallelism benefits for large machines
    - May even result in slowdowns
    - May be appropriate for small machines, but inappropriate for large ones (does not reflect real usage!)

  - Too large a problem: (chosen to be appropriate for large machine)
    - May not "fit" in small machine (thrashing to disk, key working set exceeds cache capacity, can't run at all)
    - When problem "fits" in a larger machine, super-linear speedups can occur

  - Often desirable to scale problem size as machine sizes grow (not just compute the same size of problems faster)

Scaling machines

"Does it scale?"

- Ability to scale machines is important
- Scaling up: how does its performance scale with increasing processing count?
  - Will design scale to the high end?
- Scaling down: how does its performance scale with decreasing processor count?
  - Will design scale to the low end?
- Parallel architectures designed to work in a range of contexts
  - Same architecture used, but sized differently for low end, medium scale, high end systems
  - GPUs are a great example

Questions when scaling a problem

- **Under what constraints should the problem be scaled?**
  - Fixed data set size, memory usage per processor, execution time, etc.?
  - Work may no longer be quantity that is fixed

- **How should be the problem be scaled?**

- **Problem size: defined by vector of input parameters**
  - Determines amount of work done
  - Ocean example: PROBLEM = $(n, \varepsilon, \Delta t, T)$

T: total time to simulate

Δt: time step

n: grid size

ε: convergence threshold

Scaling constraints

- User-oriented scaling properties: specific to application domain
  - Particles per processor
  - Transactions per processor

- Resource-oriented scaling properties
  1. Problem constrained scaling (PC)
  2. Memory constrained scaling (MC)
  3. Time constrained scaling (TC)

User-oriented properties often more intuitive, but resource-oriented properties are more general, apply across domains. (so we'll talk about them here)

Problem-constrained scaling

- Focus: use a parallel computer to solve the same problem faster

$$\text{Speedup} = \frac{\text{time 1 processor}}{\text{time P processors}}$$

- Recall pitfalls from earlier in lecture (small problems not realistic workloads for large machines, big problems don't fit on small machines)

- Examples:
  - Almost everything we've considered parallelizing in class so far
  - All the problems in assignment 1

Time-constrained scaling

- Focus: doing more work in a fixed amount of time
  - Execution time kept fixed as the machine (and problem) scales

$$\text{Speedup} = \frac{\text{work done by P processors}}{\text{work done by 1 processor}}$$

- How to measure "work"?
  - Execution time on a single processor? (but consider thrashing if problem too big)
  - Ideally, a measure of work is:
    - Easy to understand
    - Scales linearly with sequential complexity (So ideal speedup is linear in P)

Time-constrained scaling examples

- **Assignment 2**
  - Want real-time frame rate: ~ 30 fps
  - Faster GPU → use capability for more complex scene, not more fps

- **Computational finance**
  - Run most sophisticated model possible in: 1 hour, overnight, etc.

- **Modern web sites**
  - Want to generate complex page, respond to user in X milliseconds.

Memory-constrained scaling

- Focus: run the largest problem possible without overflowing main memory **
- Memory per processor held fixed
- Neither work or execution time are held constant

$$\text{Speedup} = \frac{\text{work(P processors) x time(1 processor)}}{\text{time(P processors) x work(1 processor)}}$$

$$= \frac{\text{work per unit time on P processors}}{\text{work per unit time on 1 processor}}$$

- Note: scaling problem size can make runtimes very large
  - Consider $O(N^3)$ matrix multiplication on $O(N^2)$ matrices

** Assumptions: (1) memory resources scale with processor count (2) spilling to disk is infeasible behavior (too slow)

Scaling examples at PIXAR

- Rendering a movie "shot" (a sequence of frames)
  - Minimizing time to completion (problem constrained)
  - Assign each frame to a different machine in the cluster

- Artists working to design lighting for a scene
  - Provide interactive frame rate in application (time constrained)
  - Buy large multi-core workstations (more performance = higher fidelity representation shown to artist)

- Physical simulation: e.g., fluids
  - Parallelize simulation across multiple machines to fit simulation in memory (memory constrained)

- Final render
  - Scene complexity bounded by memory available on farm machines
  - One barrier to exploiting additional parallelism is that footprint often increases with number of processors (memory constrained)

Case study: equation solver

- For n x n grid:
  - $O(n^2)$ memory requirement
  - $O(n^2)$ computation x number of convergence iterations = $O(n^3)$

- PC scaling:
  - Execution time: 1/P
  - Memory per processor: n^2/P
  - Concurrency: fixed at P
  - Comm-to-comp ratio: $O(P^{1/2})$

- TC scaling:
  - Let scaled grid size be k x k.
  - Assume linear speedup: $k^3/P = n^3$ (so $k = np^{1/3}$)
  - Execution time: fixed (by def.inition)
  - Memory per processor: $n^2/p^{1/3}$
  - Concurrency: $O(P^{2/3})$
  - Comm-to-comp ratio: $O(P^{1/6})$

- MC scaling:
  - Let scaled grid size by $nP^{1/2}$ x $nP^{1/2}$
  - Need $O(nP^{1/2})$ iterations to converge
  - Execution time: $O((nP^{1/2})^3/P) = O(P^{1/2})$
  - Memory per processor: fixed $O(N^2)$
  - Concurrency: O(P)
  - Comm-to-comp ratio: fixed O(1/N)

notice: execution time increases with MC scaling

Expect best speedup with MC scaling, then TC scaling (lowest overheads). Worse with PC scaling

Word of caution about problem scaling

- Problem size in the pervious examples was a single parameter n
- In practice, problem size is a combination of parameters
  - Example from Ocean: $= (n, \varepsilon, \Delta t, T)$
- Problem parameters are often related (not independent)
  - Example from Barnes-Hut: increasing star count n changes required simulation time step and force calculation accuracy parameter Θ
- Must be cognizant of these relationships when scaling problem in TC or MC scaling

Scaling summary

- Performance improvement due to parallelism is measured by speedup
- But speedup metrics take different forms for different scaling models
  - Which model matters most is application/context specific
- In addition to assignment and orchestration, behavior of a parallel program depends significantly on problem and machine scaling properties
  - When analyzing performance, be sure to analyze realistic regimes of operation (both realistic sizes and realistic problem size parameters)
  - Requires application knowledge

Back to our example of your hypothetical future job...

You have an idea for how to design a parallel machine to meet the needs of your boss.

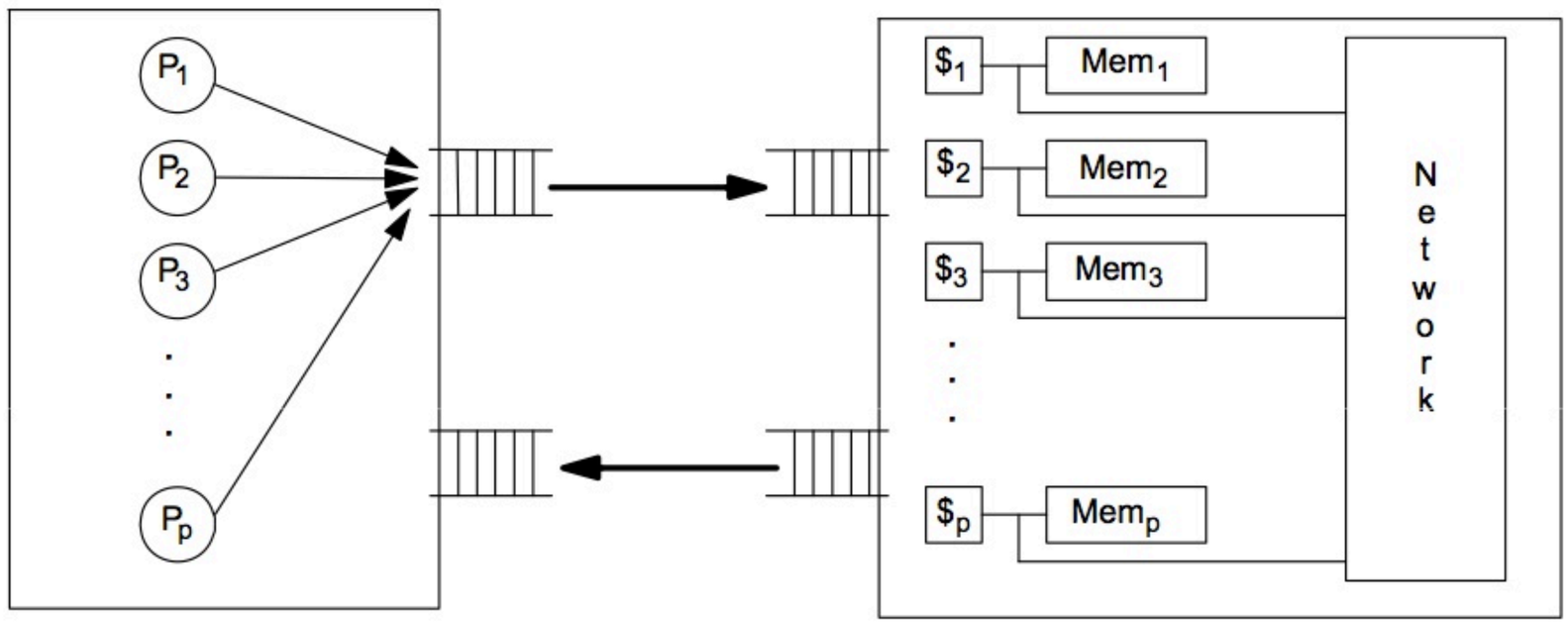
How are you going to test this idea?

Evaluating an architectural idea: simulation

- Architects evaluate architectural decisions quantitatively using simulation
  - Run with new feature, run without feature, compare simulated performance
  - Simulate against a wide collection of benchmarks

- Design detailed simulator to test new architectural feature
  - Very expensive to simulate a parallel machine in full detail
  - Often cannot simulate full machine configurations or realistic problem sizes (must scale down significantly!)
  - Architects need to be confident scaled down simulated results predict reality (otherwise, why do the evaluation at all?)

Execution-driven simulator

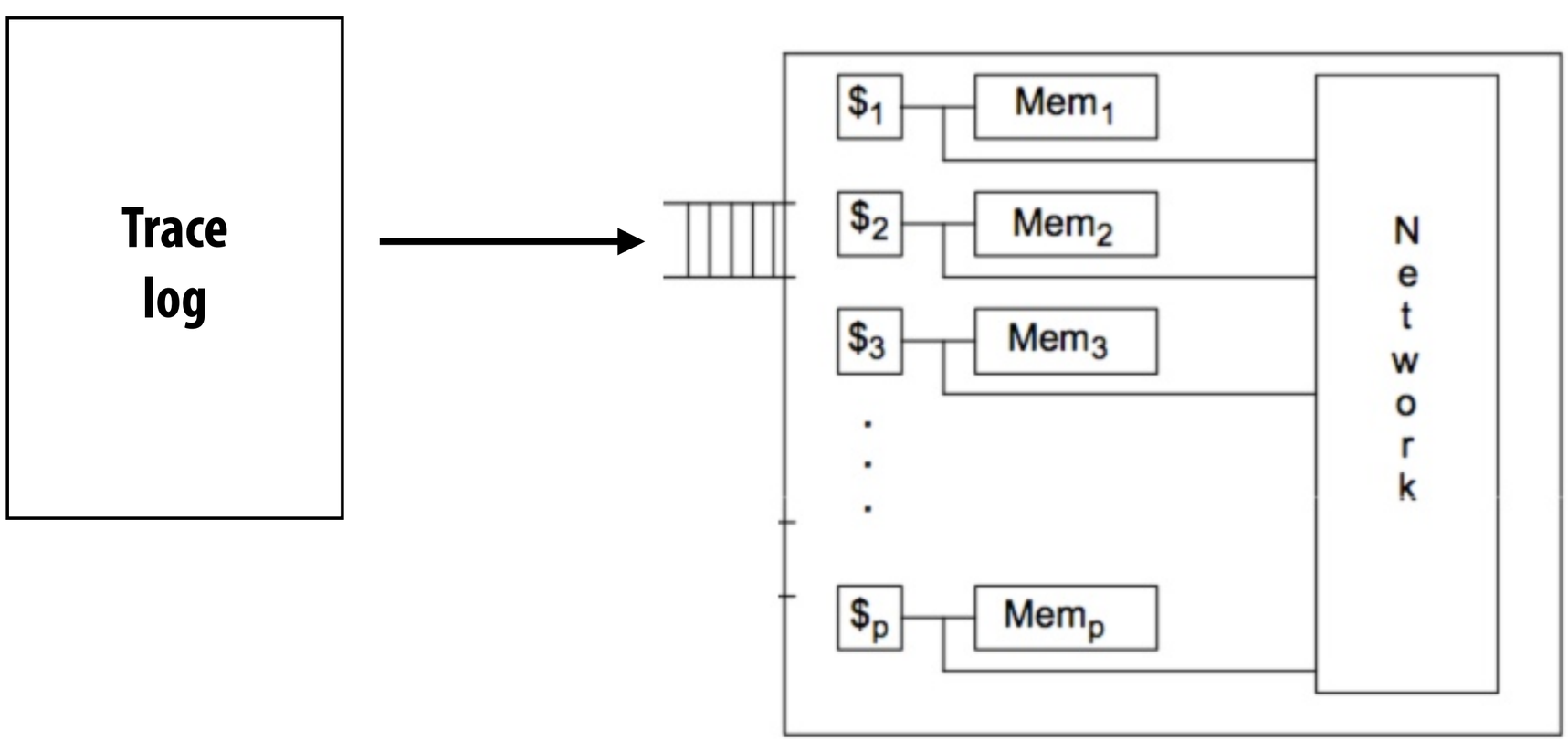


- Executes simulated program in software
  - Simulated processors generate memory references, which are processed by the simulated memory hierarchy
- Performance of simulator typically inversely proportional to level of simulated detail

Trace-driven simulator

- Instrument real code running on real machine to record a trace of all memory accesses (may also need timing info)
  - Statically (or dynamically) modify program binary
  - Example: Intel's PIN (www.pintool.org)

- Or generate trace from execution-driven simulator

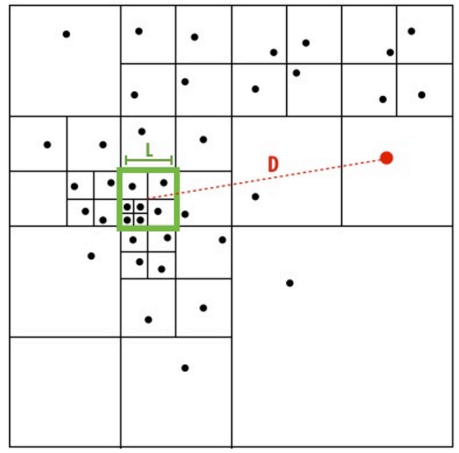


Scaling down challenges

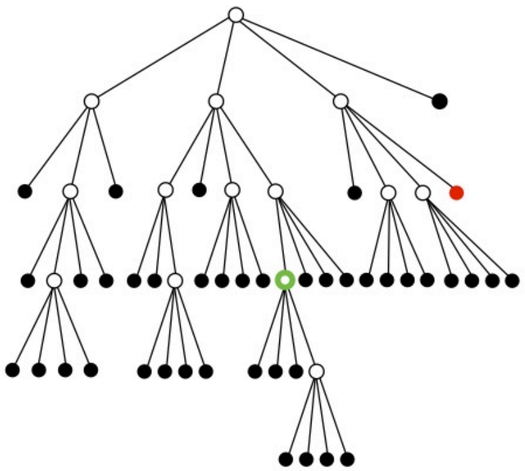
- Preserve distribution of time spent in program phases
  - e.g., Ray-trace and Barnes-Hut: both have tree build and tree traverse phases
- Preserve important behavioral characteristics
  - communication-to-computation ratio, load balance, locality, working set sizes
- Preserve contention and communication patterns
  - tough, contention is a function of timing and ratios
- Preserve scaling relationships between problem parameters
  - e.g., Barnes-Hut: scaling up particle count requires scaling down time step for physics reasons

Example: scaling down Barnes-Hut

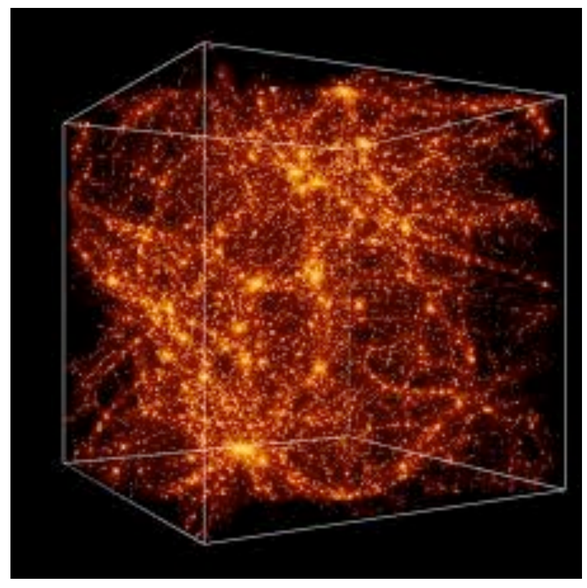
- **Problem size = (n, Θ, Δt, T)**
  - n: grid size
  - Θ: accuracy threshold
  - Δt: time step
  - T: total time to simulate



Spatial Domain


Quad-Tree Representation

- **Easiest parameter to scale down is likely T (just simulate less time)**
  - Independent of the other parameters
  - If simulation characteristics don't vary much over time
  - Or select a few representative T (beginning of sim, end of sim)

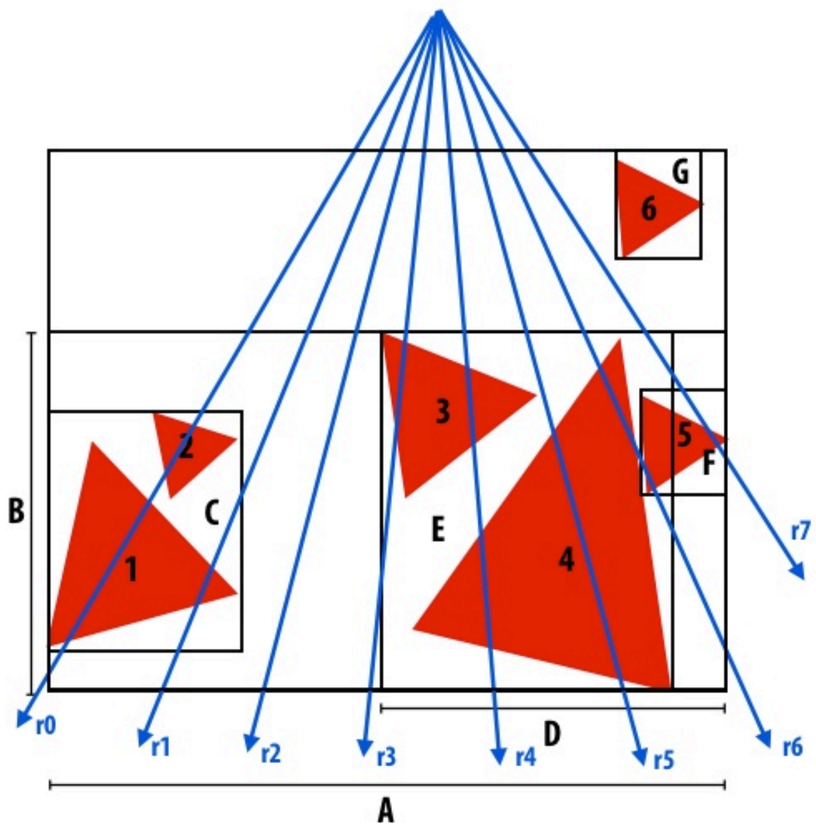


Example: scaling down the ray tracer

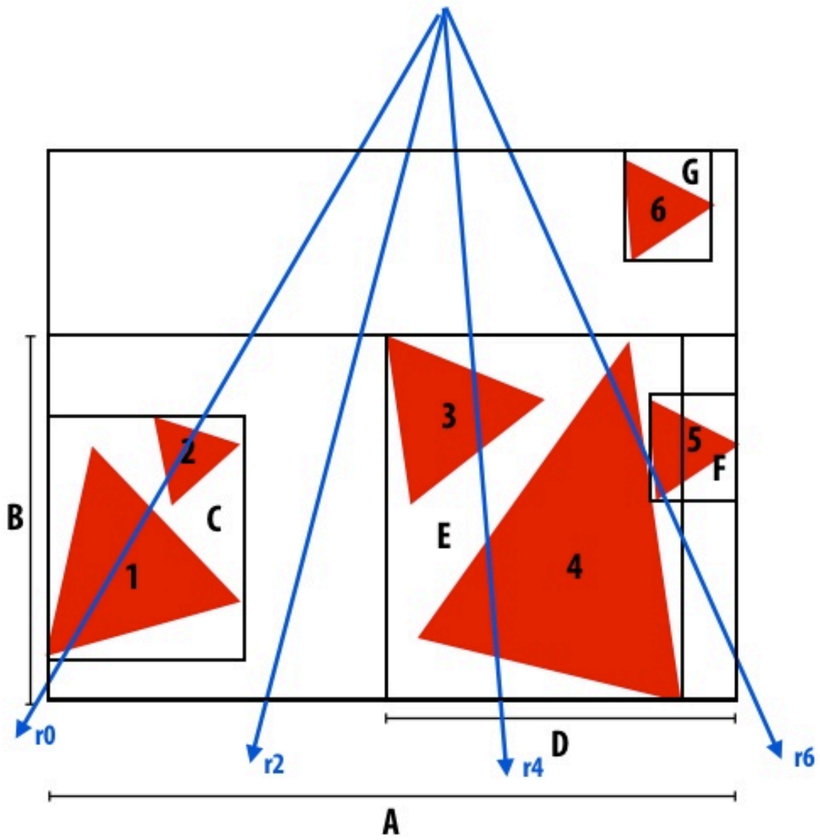
- **Problem size = (w, h, tri_count, ...)**

image width, height

number of scene triangles



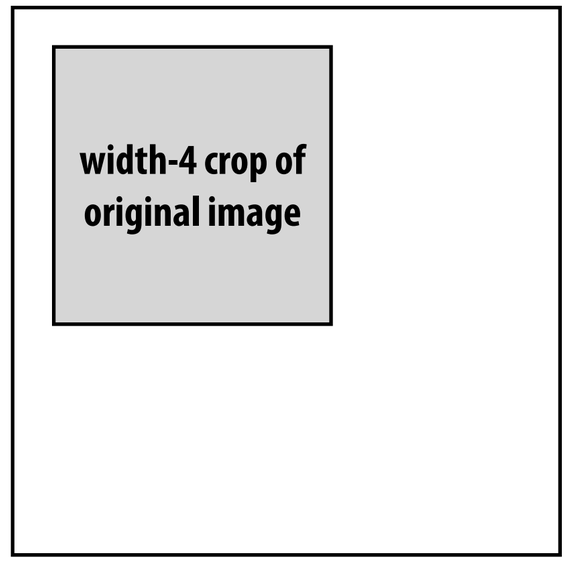
Original image
(w=8)



Smaller Image
(1/2 size: w=4)

Ray locality/divergence changes!

Better solution:
Render crop of full-size image



Original image
(w=8)

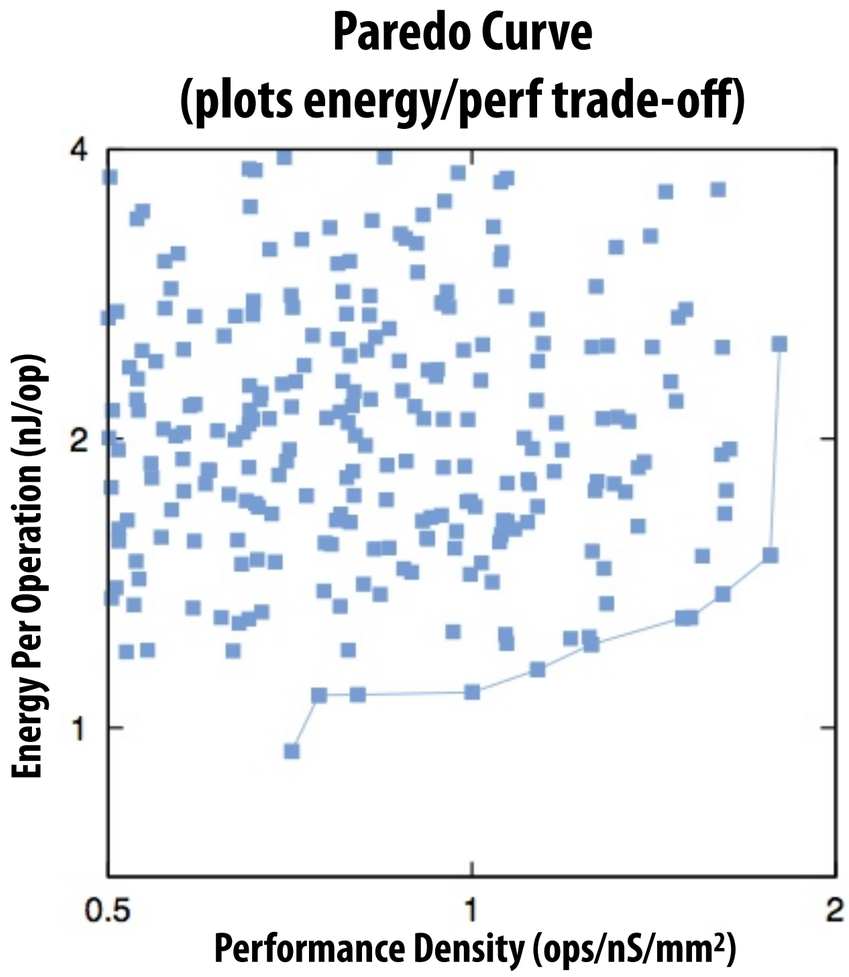
Note: issues of scaling down also apply to debugging/tuning software running on existing machines

Common example:

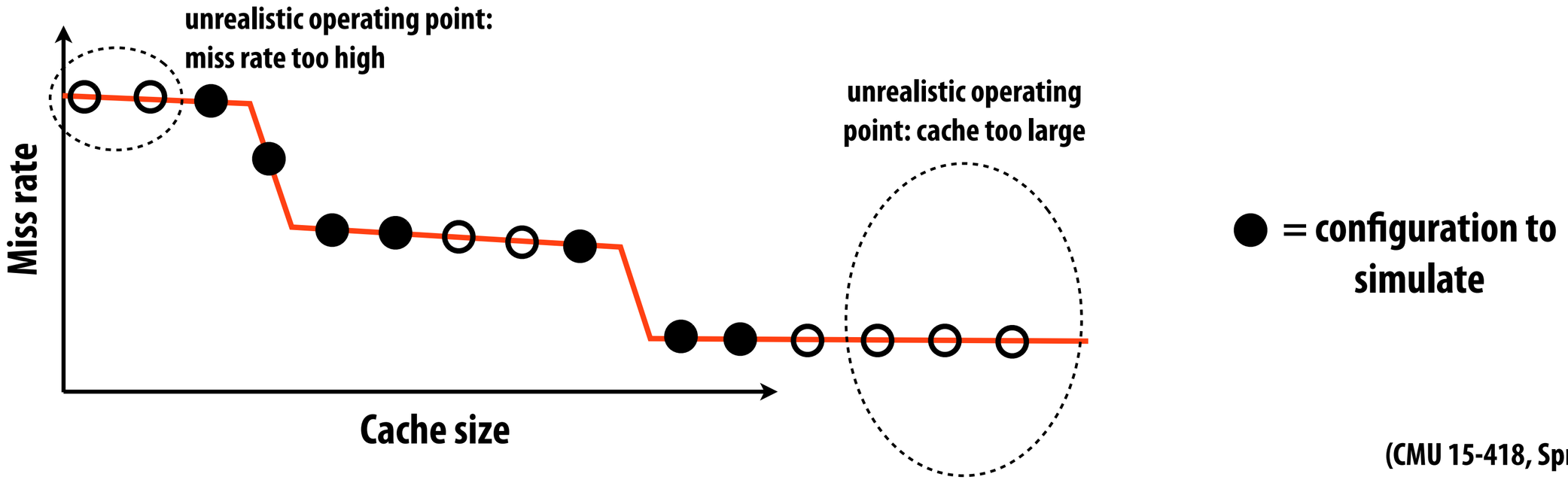
May want to log behavior of code for debugging. Instrumentation slows down code significantly, logs get untenable in size quickly.

Architectural simulation state space

- **Another evaluation challenge: dealing with large parameter space of machines**
  - Num processors, cache sizes, cache line sizes, memory bandwidths, etc.



- **Decide what parameters are relevant and prune space!**



Today's summary: BE CAREFUL!

It can be very tricky to evaluate parallel software and machines.

It can be easy to obtain misleading results.

It is helpful to precisely state your application goals. Then determine if evaluation approach is consistent with those goals.

Some tricks for evaluating the performance of software

Determine if performance is limited by computation, memory bandwidth (or latency), or synchronization **

Add math

Does runtime increase linearly with additional computation?

Remove almost all math, load same data

How much does runtime decrease? If not much, suspect bandwidth limits

Change all array accesses to A[0]

How much faster does your code get?
(this is an upper bound on benefit of exploiting locality)

Remove all atomic operations or locks

How much faster does your code get?
(this is an upper bound on benefit of reducing sync overhead)

** Often all of these effects are in play because compute, memory access, and synchronization are not perfectly overlapped. As a result, overall performance is not-dominated by exactly one type of behavior