Lecture 9:

Workload-Driven Performance Evaluation

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013

15-418 road map

- Modern multi-core chip architectures: multi-core + SIMD execution + HW multi-threading
- Ways to think about parallelism and communication
  - At the abstraction level -- programming models: shared memory, message passing, data parallelism
  - At the architectural level -- machine styles
- How to write and optimize parallel programs

  ⋮
- Case studies and example techniques

Today → **Evaluating system performance**

- Shared address space implementation details (memory coherence and consistency)

  ⋮
- Exam I

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?

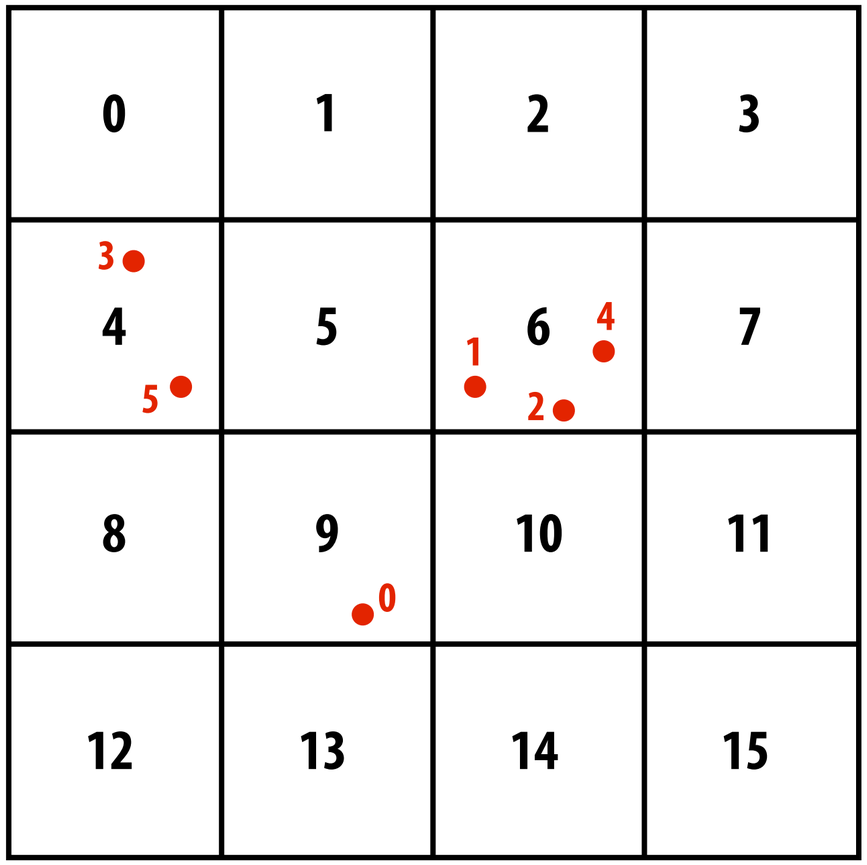
Your boss selects the application that matters most to the company
"I want you to demonstrate good performance on this application."

How do you know if you have a good design?

- **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
- **Efficiency?**
  - Performance per unit resource
  - e.g., operations per second per chip area, per dollar, per watt

Measuring scaling

- **Should speedup be measured against the parallel program running on one processor, or the best sequential program?**
  - **Recall particle binning problem from last lecture**



Parallel implementation of binning

Sequential algorithm

```
list cell_lists[16];      // 2D array of lists

for each particle p
   c = compute cell containing p
   append p to cell_lists[c]
```

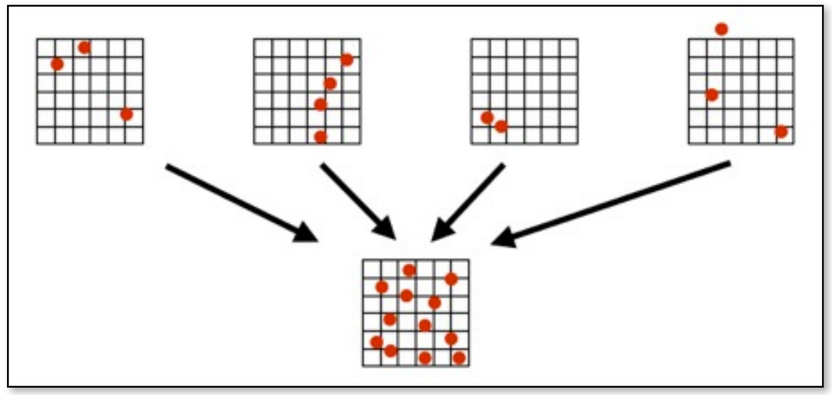
Implementation 1: Parallel over particles

```
list cell_list[16];    // 2D array of lists
lock cell_list_lock;

for each particle p    // in parallel
   c = compute cell containing p
   lock(cell_list_lock)
   append p to cell_list[c]
   unlock(cell_list_lock)
```

Implementation 2: parallel over grid cells

```
list cell_lists[16];      // 2D array of lists

for each cell c           // in parallel
   for each particle p    // sequentially
       if (p is within c)
          append p to cell_lists[c]
```

Implementation 3: build separate grids and merge



Implementation 4: data-parallel sort

Step 1: compute cell containing each particle

Array Index:	0	1	2	3	4	5
result:	9	6	6	4	6	4

Step 2: sort results by cell

Array Index:	3	5	1	2	4	0
result:	4	4	6	6	6	9

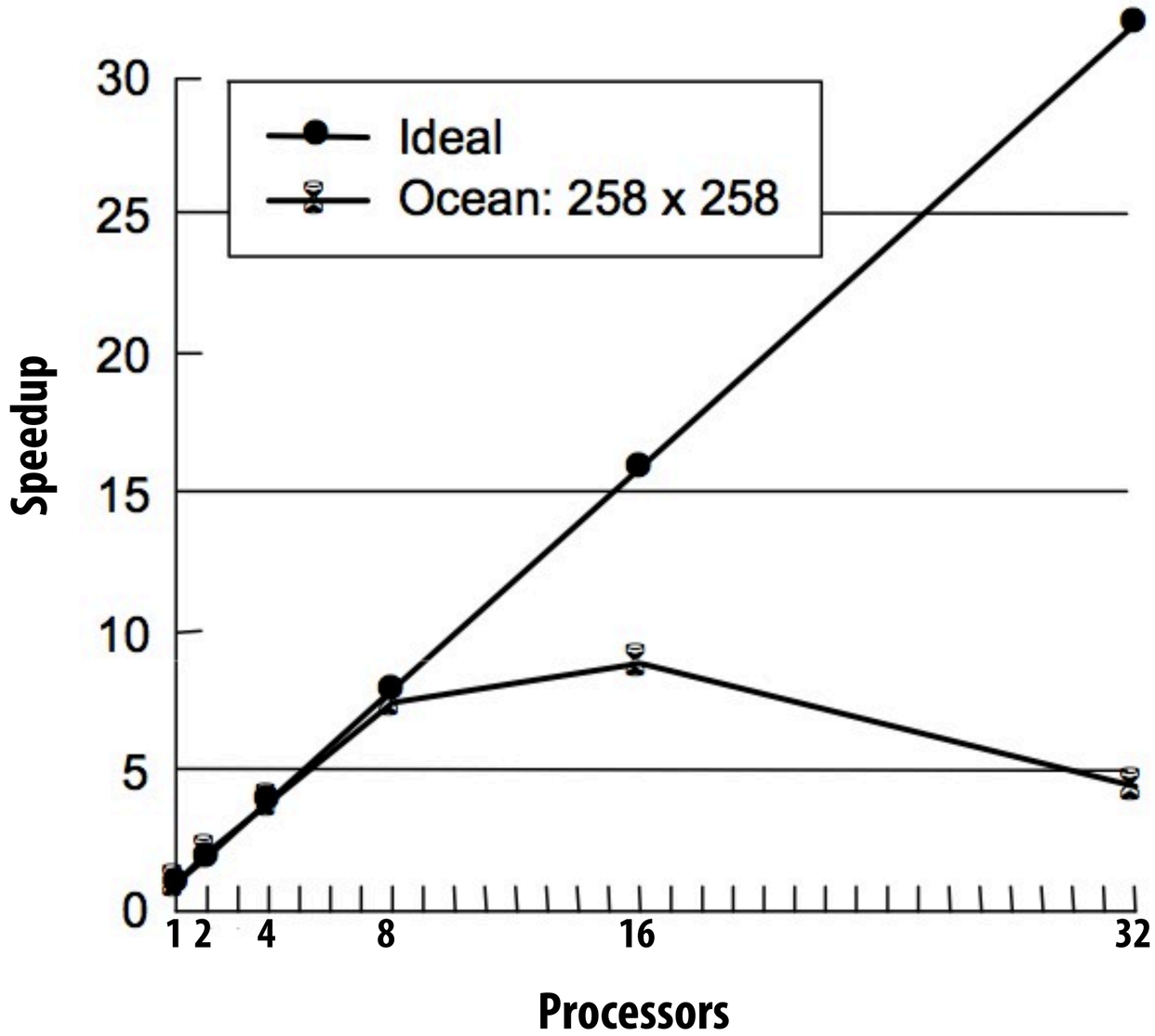
Step 3: find start/end of each cell

```
cell = result[index]
if (index == 0 || cell != result[index-1]) {
    cell_starts[cell] = index;
    if (index > 0)           // special case for first cell
       cell_ends[result[index-1]] = index;
}
if (index == numParticles-1) // special case for last cell
    cell_ends[cell] = index+1;
```

Common pitfall: compare parallel program speedup to parallel program on one core (easier to make yourself look good)

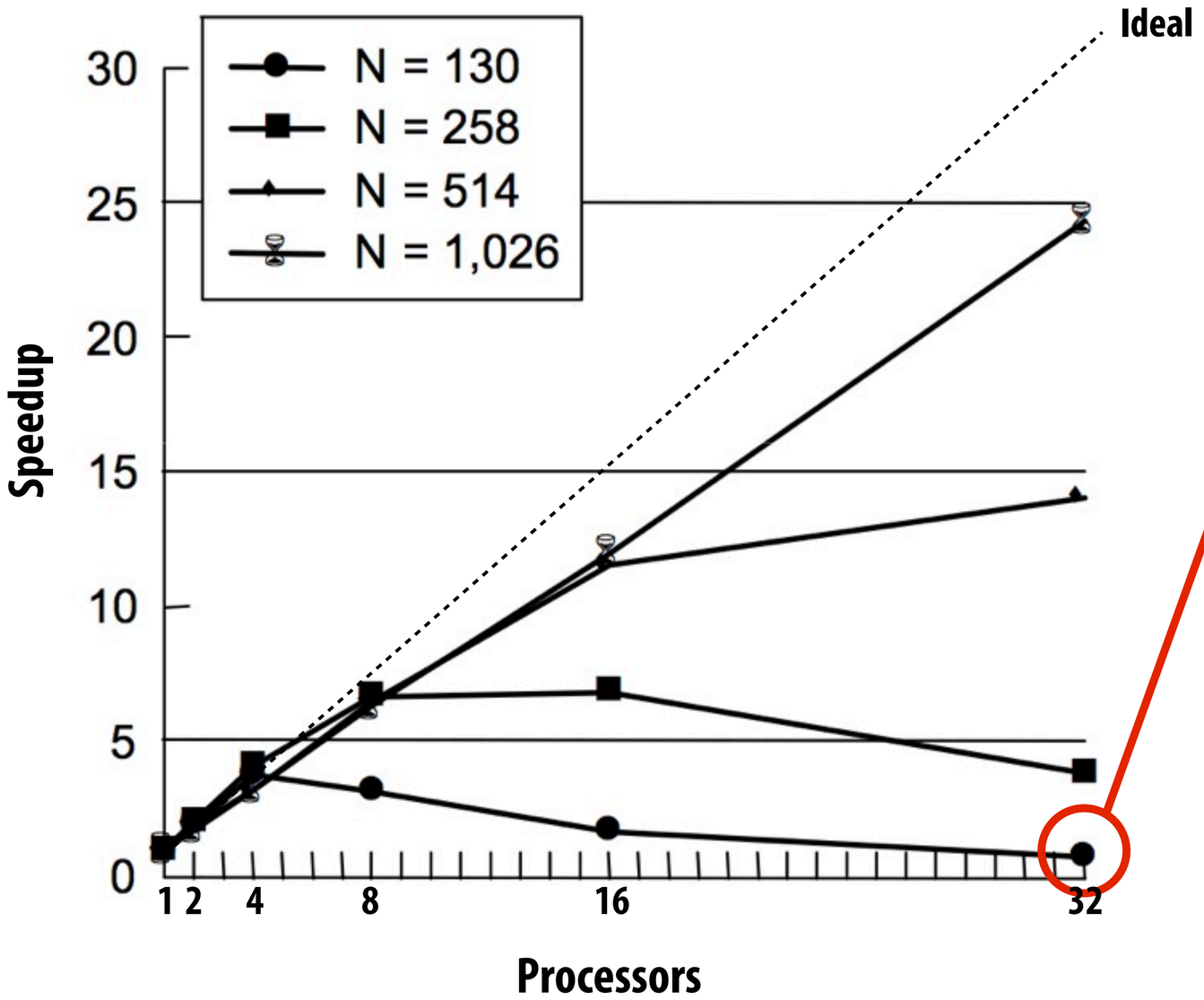
Speedup of ocean sim application: 258 x 258 grid

Execution on 32 processor SGI Origin 2000



Pitfalls of fixed problem size speedup analysis

Ocean execution on 32 processor SGI Origin 2000



No benefit! (slight slowdown)

Problem is just too small for the machine (large communication-to-computation ratio)

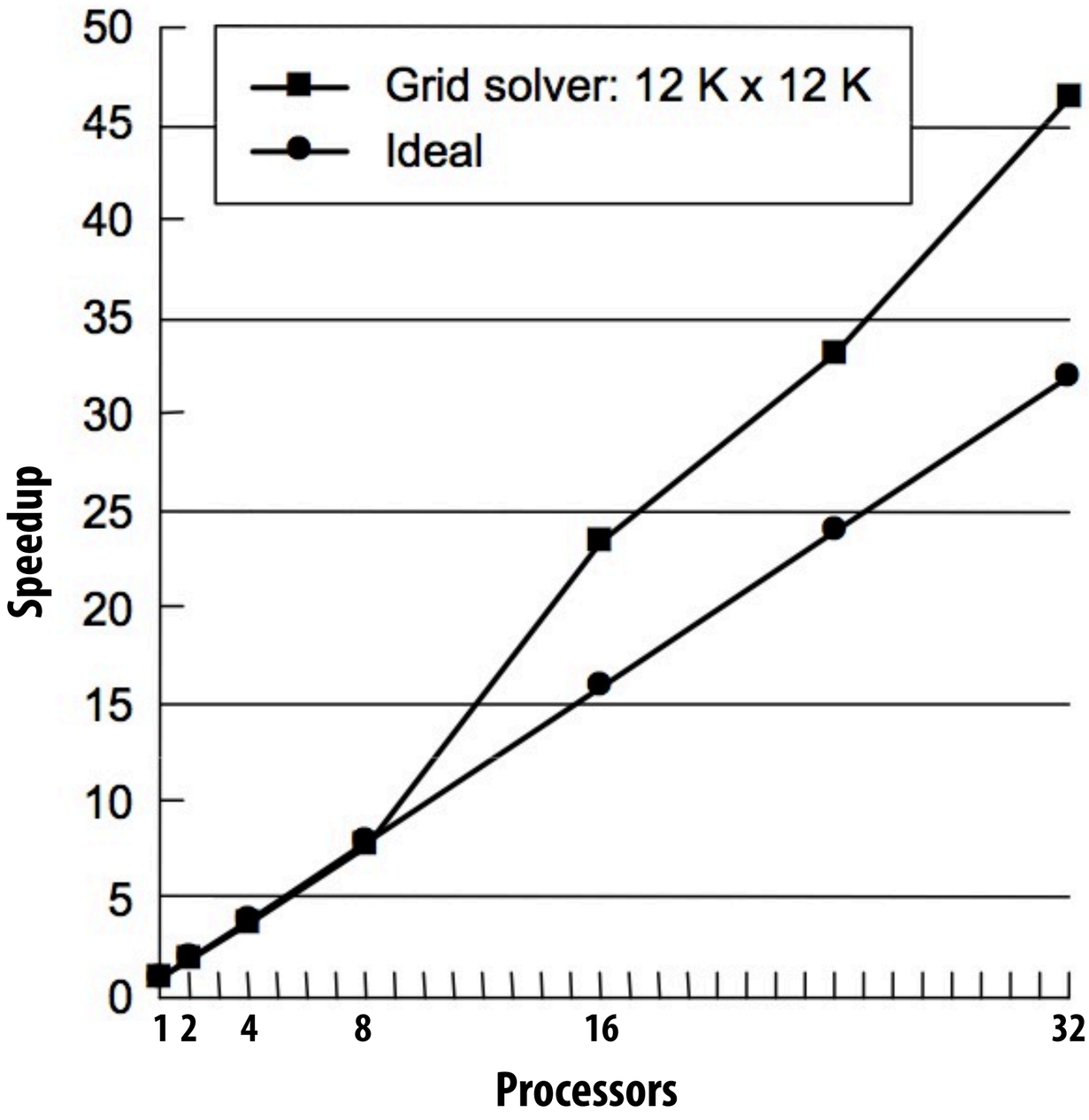
Scaling the performance of small problem may not matter anyway (it could execute fast enough on a uniprocessor)

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



Here: super-linear speedup! with enough processors, key working set fits in per-processor cache

Another example: if problem size is too large for a single small machine, working set may not fit in memory: causing thrashing 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 of large machine!)

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

  - Can be desirable to scale problem size as machine sizes grow (rather than just compute the same size of problems faster)

Scaling machines

"Does it scale?"

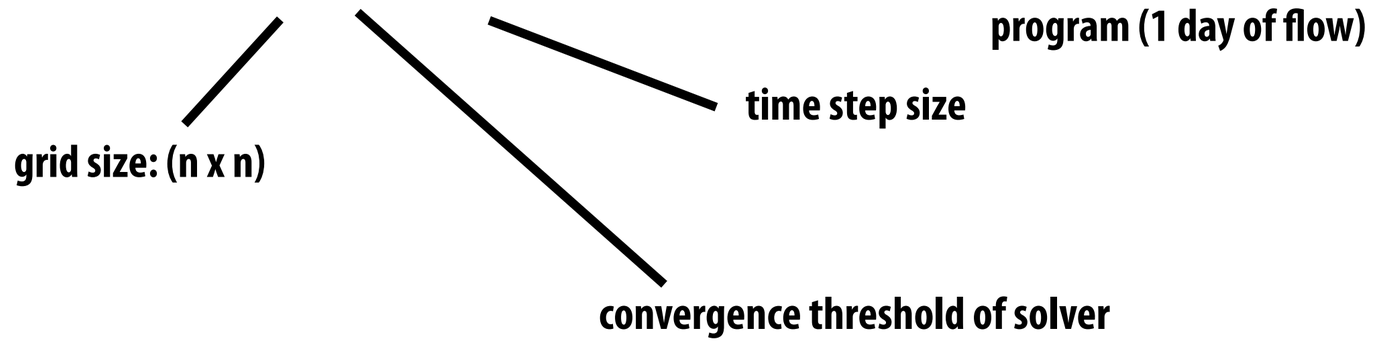
- Ability to scale machines (or software parallelization strategies) is important
- Scaling up: how does its performance scale with increasing processing count?
  - Will design scale to the high end?
- Scaling down: how does its performance scale with decreasing processor count?
  - Will design scale to the low end?
- Parallel architectures are designed to work in a range of contexts
  - Same architecture used for low-end, medium-scale, and high-end systems
  - GPUs are a great example (e.g., GTX 650, GTX 670, and GTX 680 in the lab)

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: often determined by more than one parameter
  - Determines amount of work done
  - Ocean example: PROBLEM = (n, ε, Δt, T)



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 of problem-constrained scaling:
  - 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"?
  - "Work done" often not linear in values of problem inputs
  - 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 run time (So ideal speedup remains linear in P)

Time-constrained scaling examples

- **Assignment 2 (real-time graphics)**
  - Want real-time frame rate: ~ 30 fps
  - Faster GPU → use capability for more complex scene, not to render more frames per second

- **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 (because studies show usage directly corresponds to page load latency)

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 "shot" (a sequence of frames) in a movie
  - 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)
  - More performance = higher fidelity representation shown to artist in allotted time

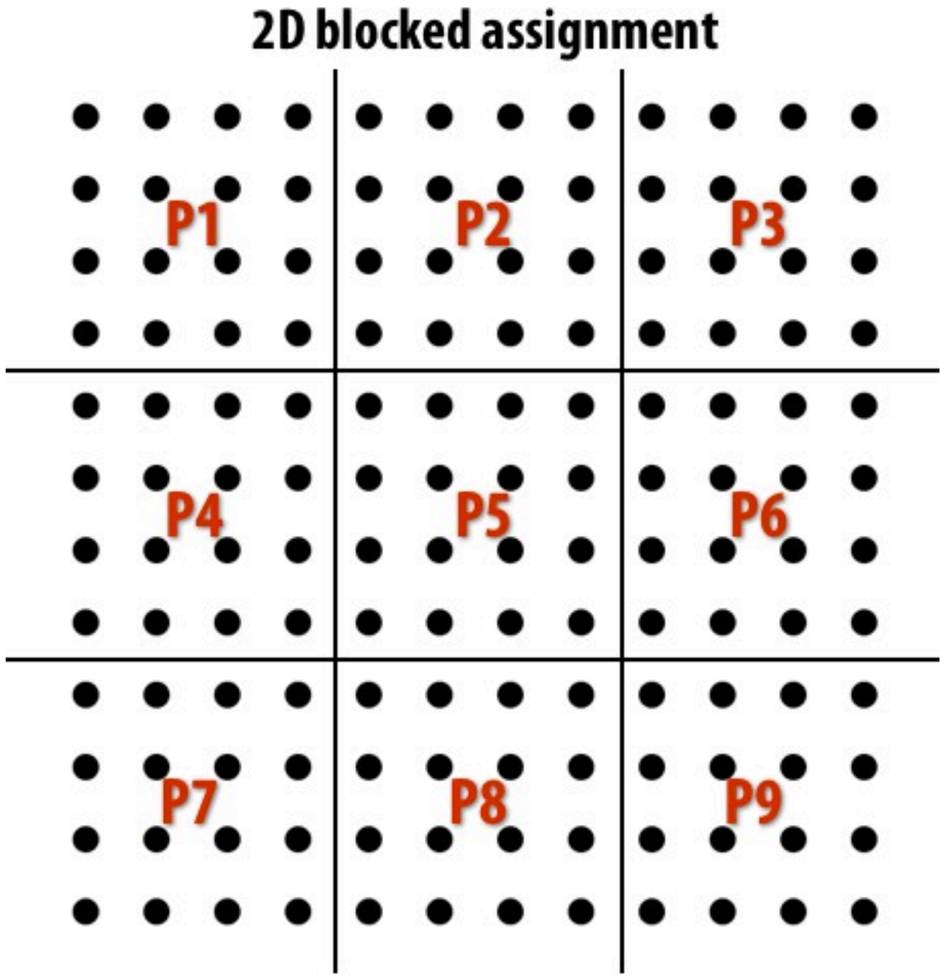
- Physical simulation: e.g., fluids
  - Parallelize simulation across multiple machines to fit simulation in memory (memory constrained)

- Final render of images for movie
  - 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: our 2D grid solver

- For n x n grid:
  - $O(n^2)$ memory requirement
  - $O(n^2)$ grid elements x $O(n)$ convergence iterations... so total work increases as $O(n^3)$

- Problem-constrained scaling:
  - Execution time: 1/P
  - Memory per processor: n^2/P
  - Concurrency: fixed at P
  - Comm-to-comp ratio: $O(P^{1/2})$
    (for a 2D-blocked assignment)



N^2 elements

P processors

elements computed: $\frac{N^2}{P}$
(per processor)

elements communicated: $\propto \frac{N}{\sqrt{P}}$
(per processor)

Case study: our 2D grid solver

- For n x n grid:
  - $O(n^2)$ memory requirement
  - $O(n^2)$ grid elements x $O(n)$ convergence iterations... so total work increases as $O(n^3)$

notice: execution time increases with MC scaling

- Problem-constrained scaling:
  - Execution time: 1/P
  - Memory per processor: n^2/P
  - Comm-to-comp ratio: $O(P^{1/2})$
    (for a 2D-blocked assignment)

- Memory-constrained 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 $O(N^2)$ (by definition)
  - Comm-to-comp ratio: $O(1/(NP^{1/2}))$ (it decreases!)

- Time-constrained scaling:
  - Let scaled grid size be k x k
  - Assume linear speedup: $k^3/P = n^3$ (so $k = np^{1/3}$)
    (computation time for k x k grid on P processors = computation time for n x n grid on 1 processor)
  - Execution time: fixed at $O(n^3)$ (by defn of TC scaling)
  - Memory per processor: $k^2/p = n^2/p^{1/3}$
  - Comm-to-comp ratio: $O(P^{1/6})$

Implications:
Expect best "speedup" with MC scaling, then TC scaling, worst with PC scaling.
(due to communication overheads)

Word of caution about problem scaling

- Problem size in the previous examples was a single parameter n
- In practice, problem size is a combination of parameters
  - Prior example from Ocean: = (n, ε, Δt, T)
- Problem parameters are often related (not independent)
  - Example from Barnes-Hut: increasing particle 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.

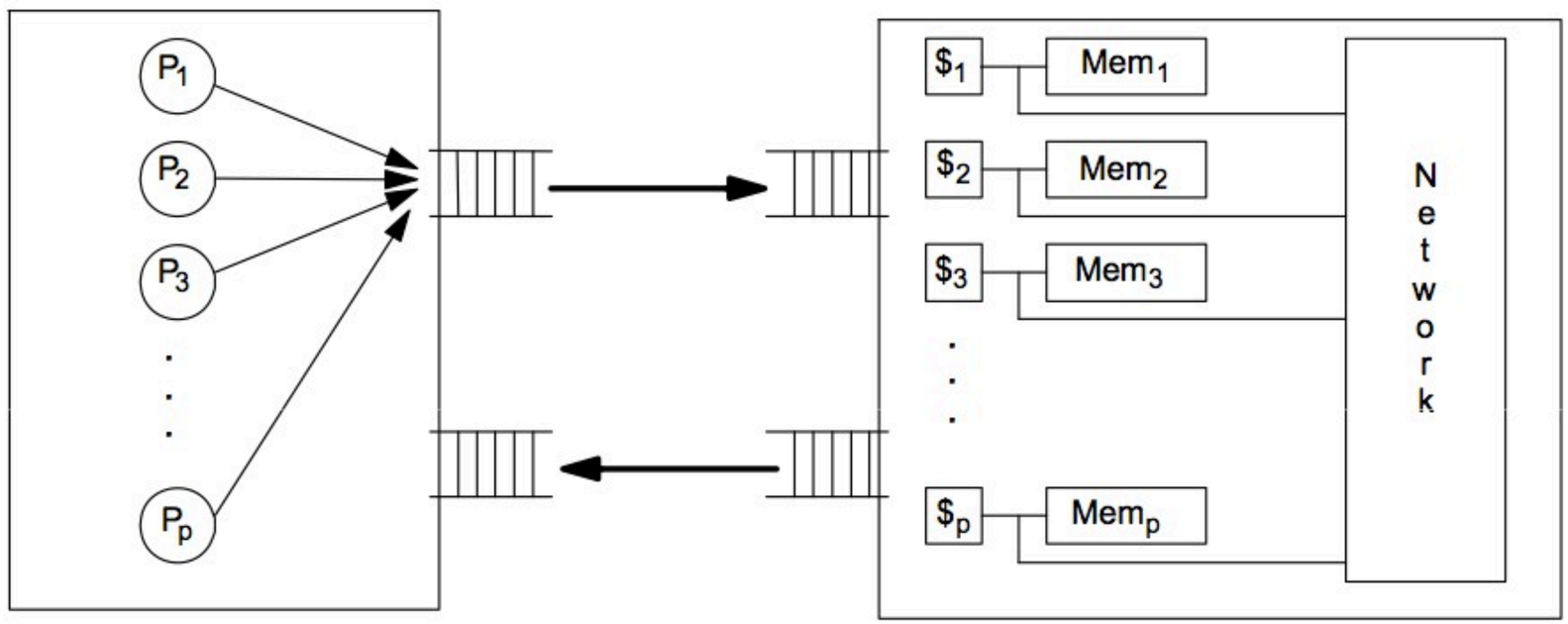
How are you going to test this idea?

Evaluating an architectural idea: simulation

- Architects evaluate architectural decisions quantitatively using chip simulators
  - 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 workloads significantly!)
  - Architects need to be confident scaled down simulated results predict reality (otherwise, why do the evaluation at all?)

Execution-driven simulator



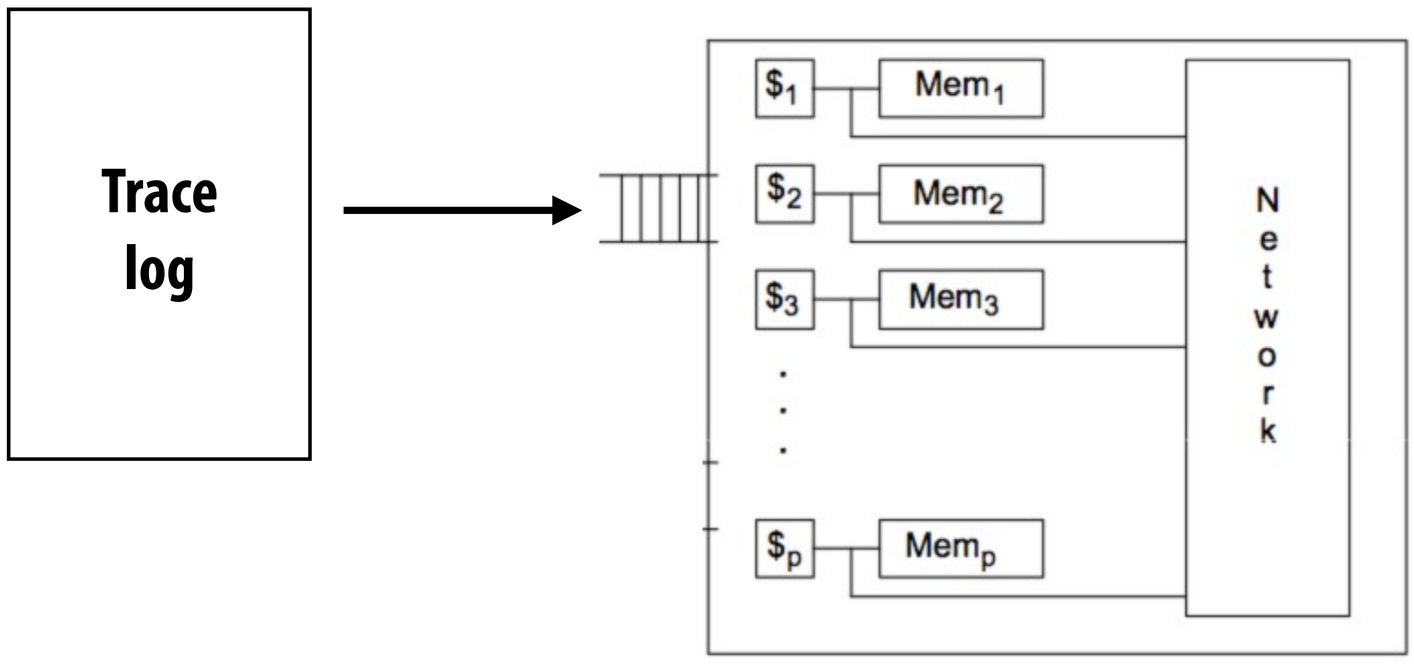
- Executes simulated program in software
  - Simulated processors generate memory references, which are processed by the simulated memory hierarchy

- Performance of simulator typically inversely proportional to level of simulated detail
  - Simulate every instruction? Simulate every bit on every wire?

Trace-driven simulator

- Instrument real code running on real machine to record a <u>trace</u> of all memory accesses
  - Statically (or dynamically) modify program binary
  - Example: Intel's PIN (<u>www.pintool.org</u>)
  - May also need to record timing information to model contention in subsequent simulation

- Or generate trace using an execution-driven simulator
- Then play back trace on simulator



Scaling down (or up) challenges

- Preserve ratio of time spent in program phases
  - e.g., Ray-trace and Barnes-Hut: both have tree build and tree traverse phases
  - Shrinking screen size changes cost of tracing rays but not cost of building tree
  - Changing density of particles changes per particle cost in Barnes-Hut

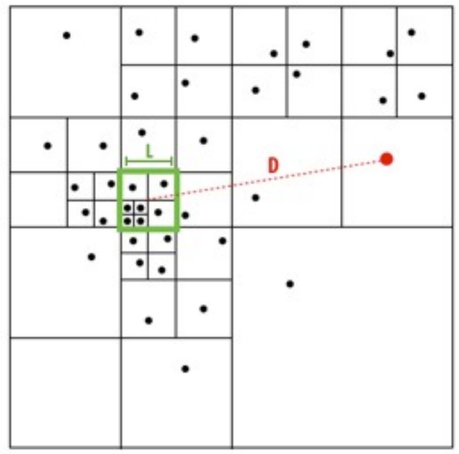
- Preserve important behavioral characteristics
  - Communication-to-computation ratio, load balance, locality, working set sizes
  - e.g., shrinking grid size in solver changes communication to computation ratio

- Preserve contention and communication patterns
  - Tough to preserve contention since contention is a function of timing and ratios

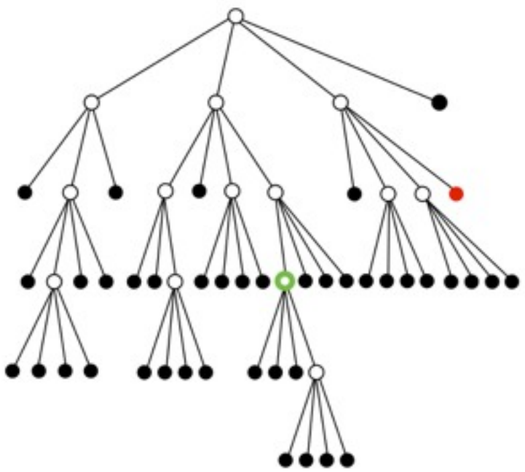
- Preserve scaling relationships between problem parameters
  - e.g., Barnes-Hut: scaling up particle count requires scaling down time step for physics reasons

Example: scaling down Barnes-Hut

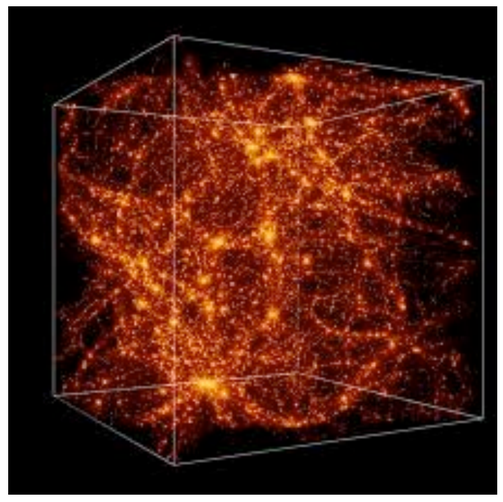
- **Problem size = (n, Θ, Δt, T)**
  - n: grid size
  - Θ: accuracy threshold
  - Δt: time step
  - T: total time to simulate



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 (but they probably do: gravity bodies particles together over time and performance depends on density)
  - On solution: select a few representative periods of time (e.g., sequence of time steps at beginning of sim, at end of sim)

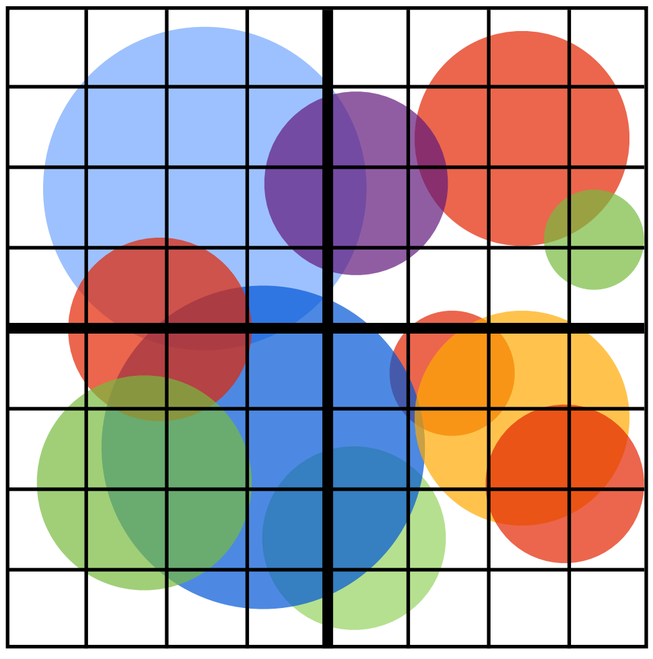


Example: scaling down assignment 2

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

image width, height

number of scene circles



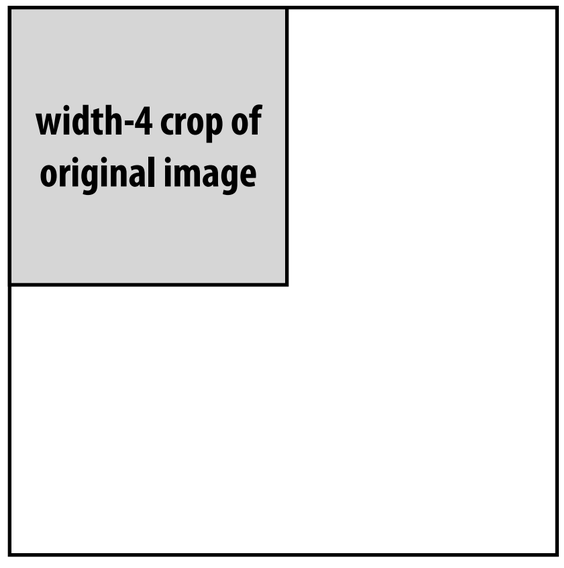
Original image
(w=8)

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

Shrink image, but keep circle data the same:

- Ratio of "filtering" work to per-pixel work changes
- Concentration of circles in a tile changes

Alternative solution:
Render crop of full-size image



Original image
(w=8)

Note: issues of scaling down also apply to software developers debugging/tuning parallel programs running on real machines

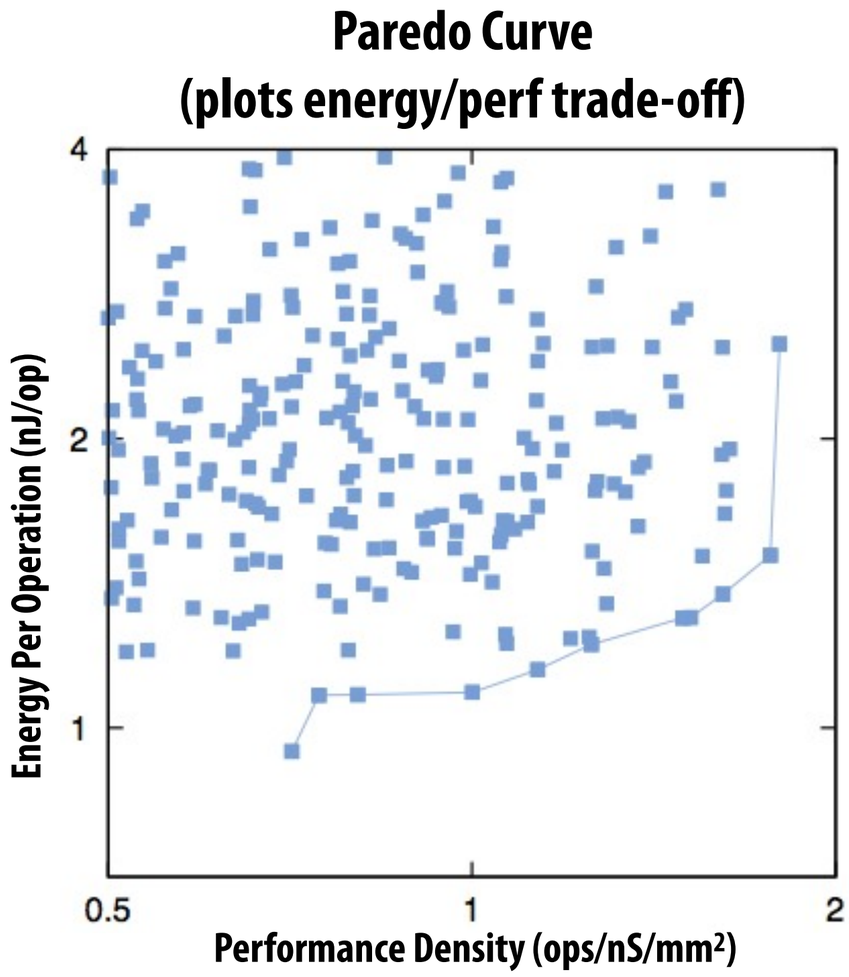
Common examples:

- **May want to log behavior of code for debugging**
  - **Instrumentation slows down code significantly**
  - **Debug logs get untenable in size quickly**

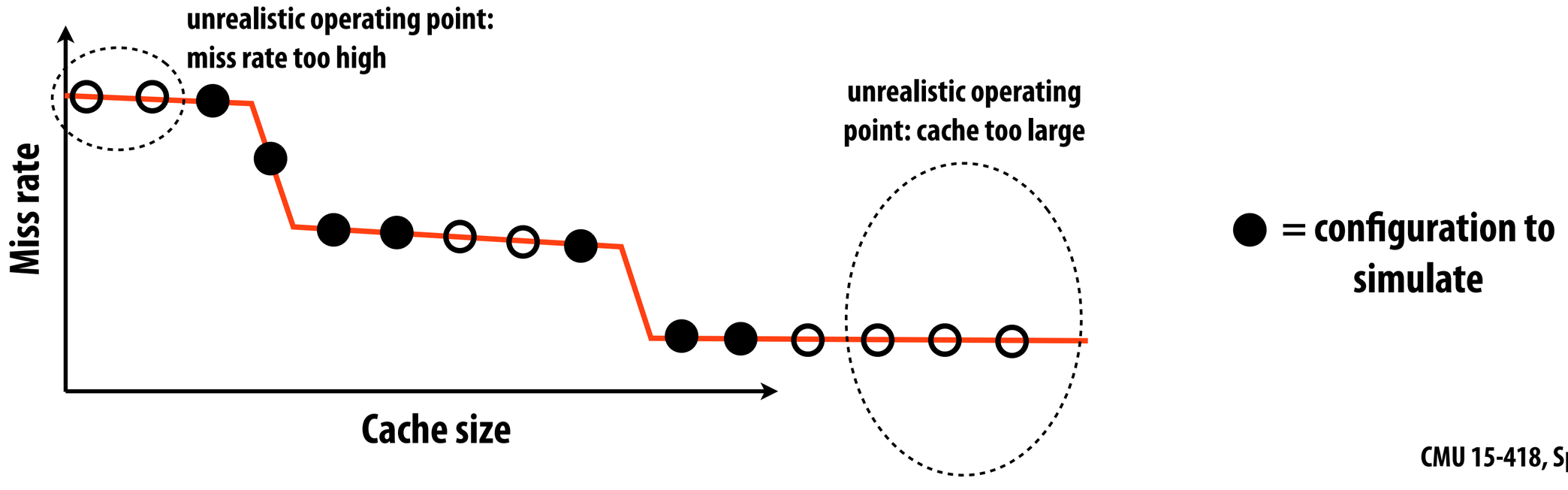
- **May want to debug/test on small problem on small machine before running two-week simulation on full problem size on a supercomputer**

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 and tune parallel software and machines.

It can be very easy to obtain misleading results or tune your code (or hardware design) for a workload that is not representative of real-world use cases.

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

Bonus material: some tricks for understanding the performance of parallel software

Key ideas

- Determine if your performance is limited by computation, memory bandwidth (or memory latency), or synchronization?

- Try and establish "high watermarks"
  - What's the best you can do?
  - How close is your implementation to a best case scenario?

Establishing high watermarks **

Add "math" (non-memory instructions)

Does runtime increase linearly with additional computation? (Signs point to instruction-rate limited)

Remove almost all math, but load same data

How much does runtime decrease? If not much, suspect memory bottleneck

Change all array accesses to A[0]

How much faster does your code get?

(This establishes an upper bound on benefit of exploiting locality)

Remove all atomic operations or locks

How much faster does your code get? (provided it still does approximately the same amount of work)

(This establishes an upper bound on benefit of reducing sync overhead.)

** Often all of these effects are in play because computation, memory access, and synchronization are never perfectly overlapped. As a result, overall performance will rarely be dictated by exactly one type of behavior. But the amount performance changes as a result of each of these tests can be a good indication of dominant costs