Lecture 10:

Workload-Driven Performance Evaluation

Parallel Computer Architecture and Programming

CMU 15-418/15-618, Spring 2019

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

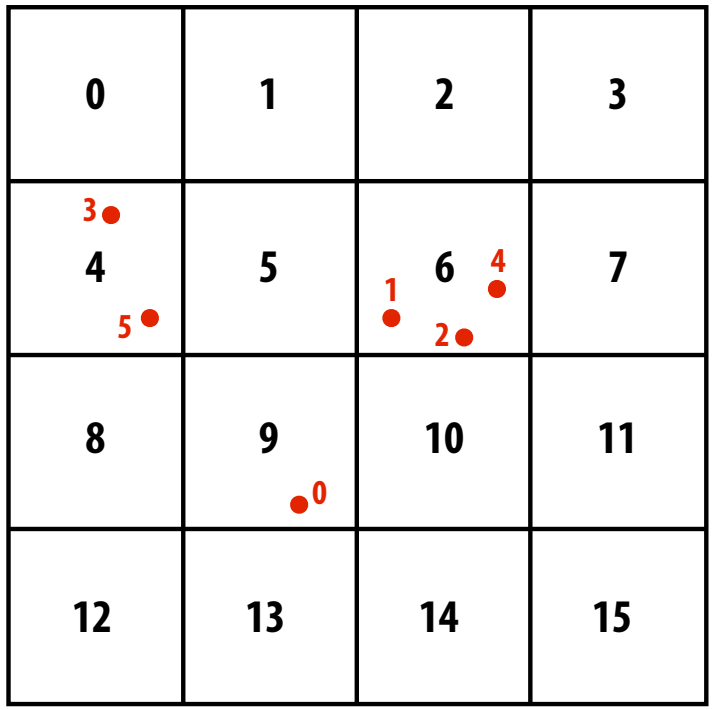
"I want you to demonstrate good performance on this application."

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

- Consider the particle binning example from last week's class
- Should speedup be measured against the performance of a parallel version of a program running on one processor, or the best sequential program?



Parallel implementations of particle 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]
```

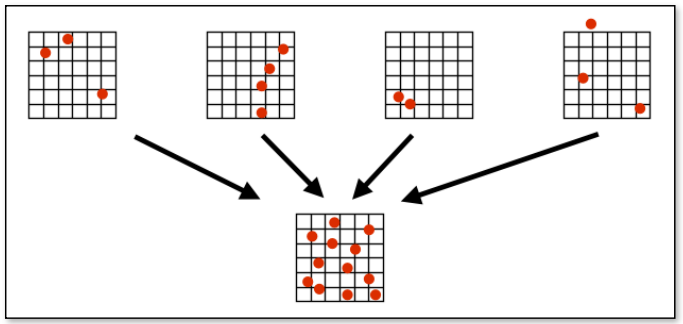
Implementation 1: Parallel over particles

```
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 algorithm running 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

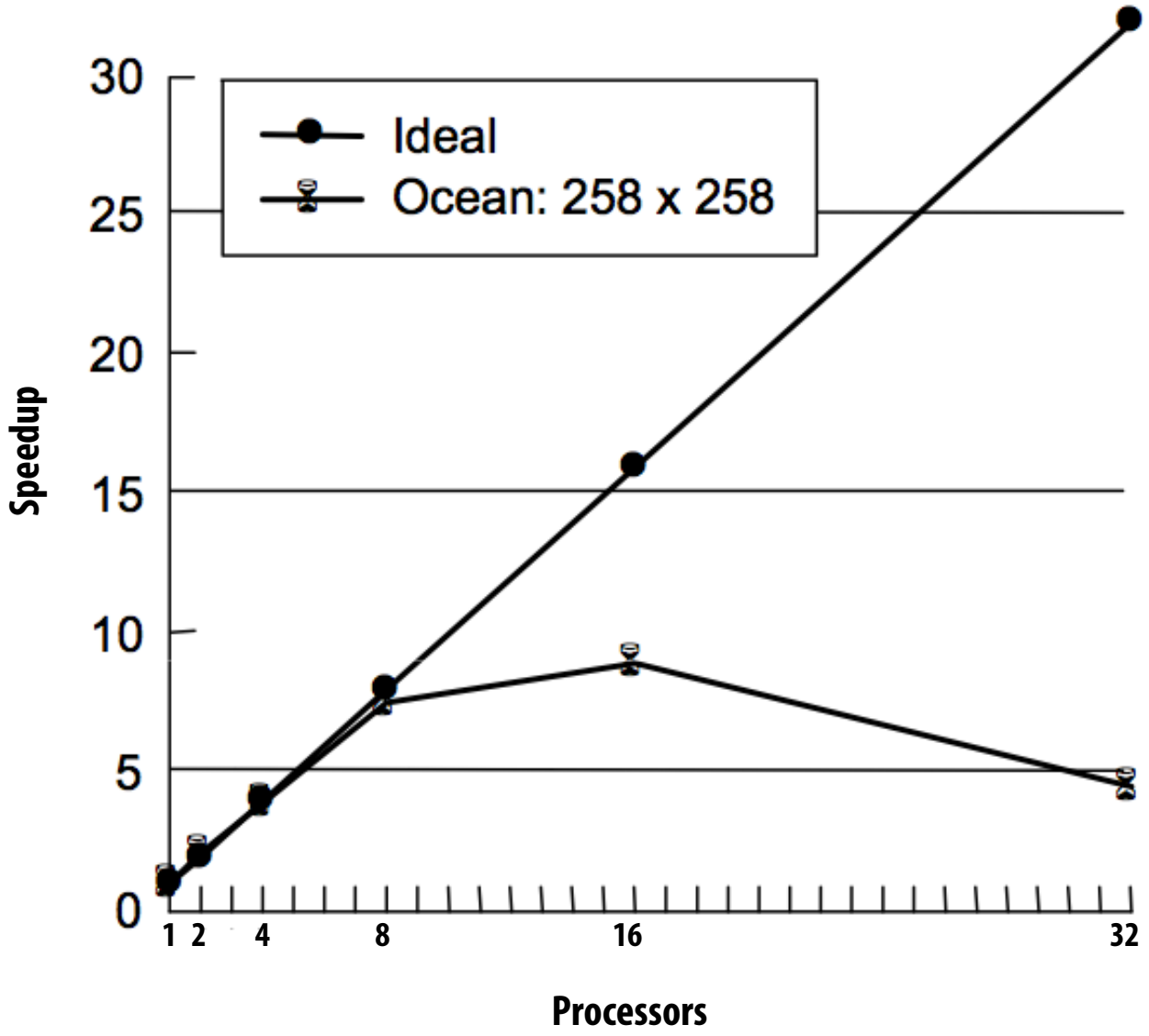
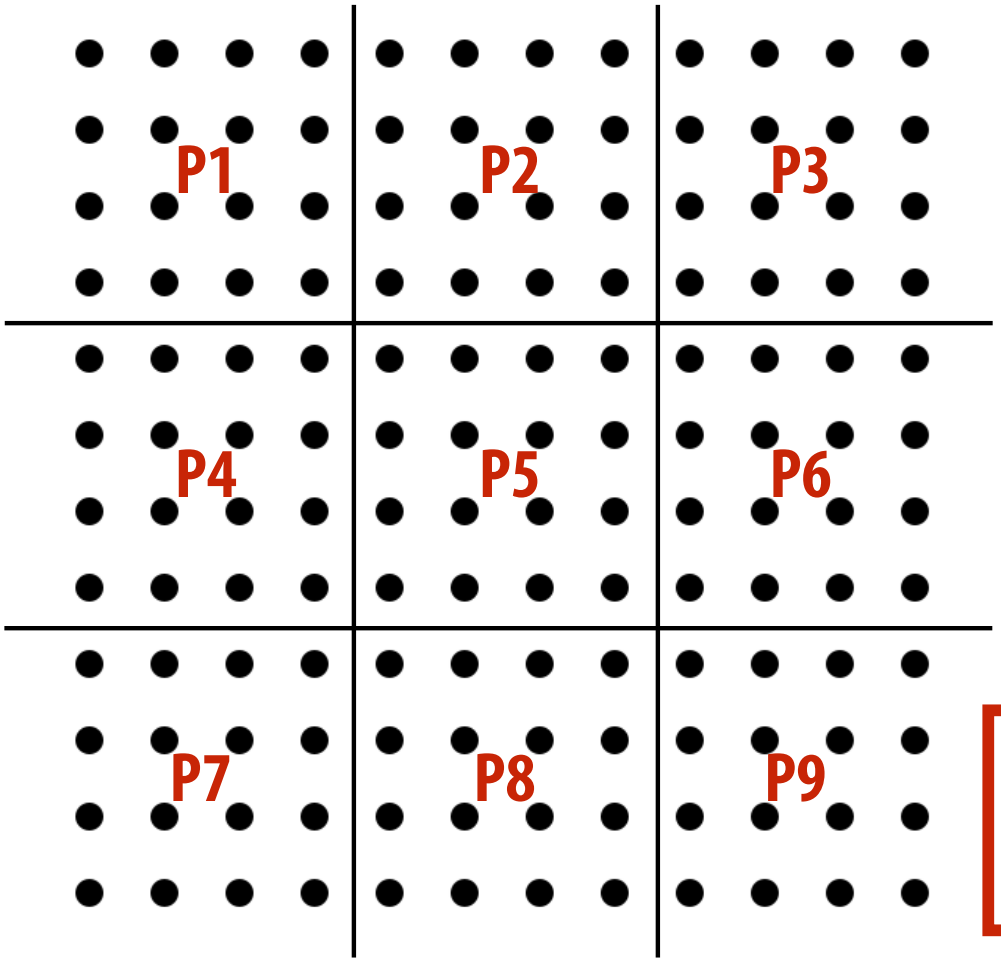


Figure credit: Culler, Singh, and Gupta

Remember: work assignment in ocean

2D blocked assignment: N x N grid



N^2 elements

P processors

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

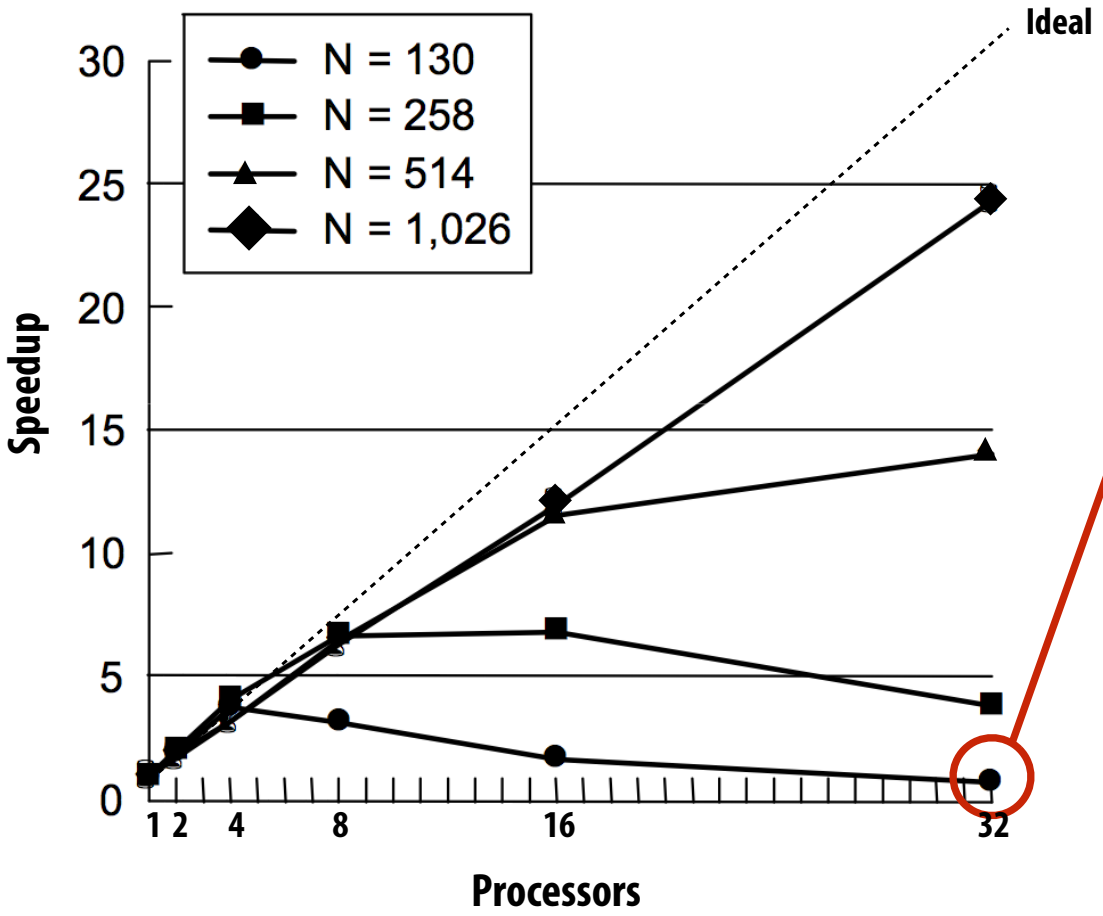
elements communicated: (per processor) $\propto \frac{N}{\sqrt{P}}$

arithmetic intensity: $\frac{N}{\sqrt{P}}$

Small N yields low arithmetic intensity!

Pitfalls of fixed problem size speedup analysis

Ocean execution on 32 processor SGI Origin 2000



No benefit! (slight slowdown)

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

Scaling the performance of small problem may not be all that important anyway (it might already execute fast enough on a single core)

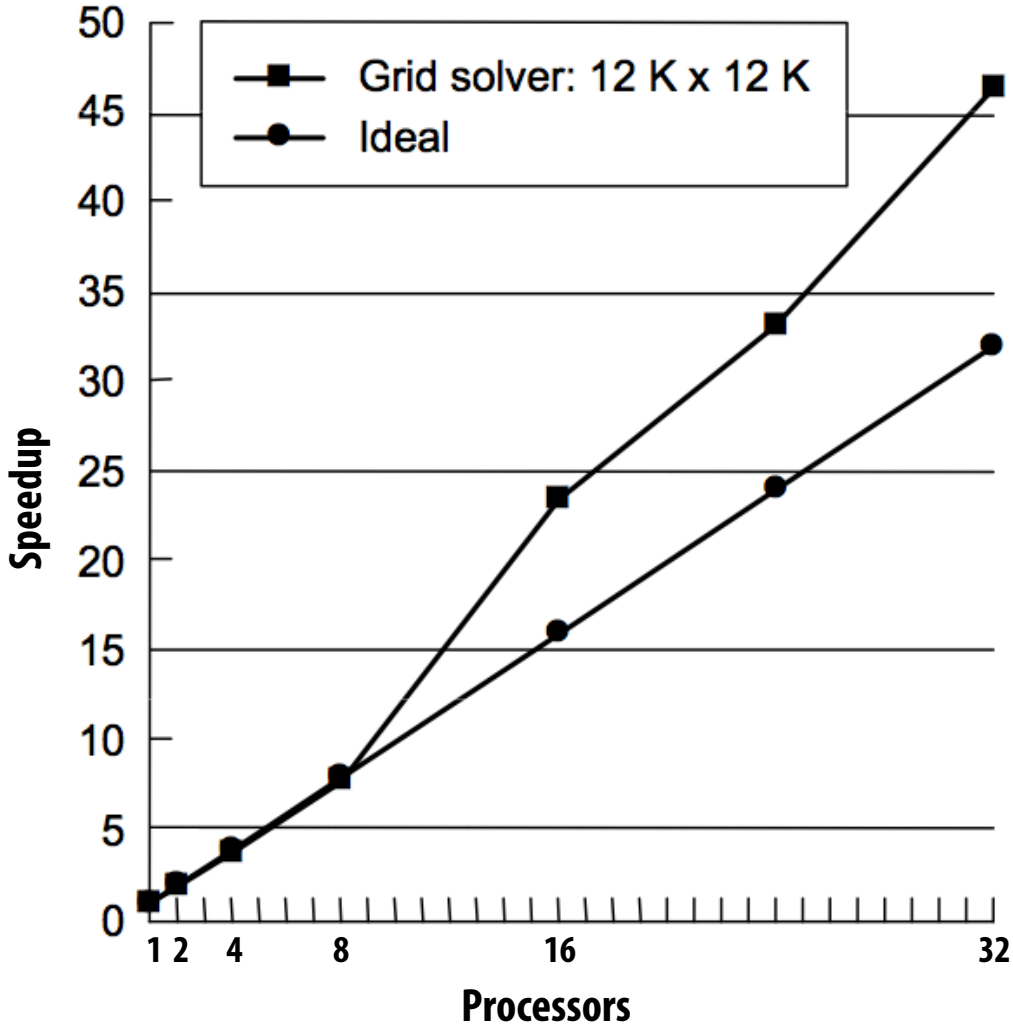
258 x 258 grid on 32 processors: ~ 310 grid cells per processor

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

Figure credit: Culler, Singh, and Gupta

Pitfalls of fixed problem size speedup analysis

Execution on 32 processor SGI Origin 2000



Here: super-linear speedup! with enough processors, chunk of grid assigned to each processor begins to fit in cache (key working set fits in per-processor cache)

Another example: if problem size is too large for a single machine, working set may not fit in memory: causing thrashing to disk

(this would make speedup on a bigger parallel machine with more memory look amazing!)

Figure credit: Culler, Singh, and Gupta

Understanding scaling: size matters!

- There can be complex interactions between the size of the problem and the size of the parallel computer.
  - Can impact load balance, overhead, arithmetic intensity, locality of data access
  - Effects can be dramatic and application dependent

- Evaluating a machine with a fixed problem size can be problematic
  - Too small a problem:
    - Parallelism overheads dominate parallelism benefits (may even result in slow downs)
    - Problem size may be appropriate for small machines, but inappropriate for large ones (does not reflect realistic usage of large machine!)
  - Too large a problem: (problem size chosen to be appropriate for large machine)
    - Key working set 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 working set "fits" in a large machine but not small one, super-linear speedups can occur

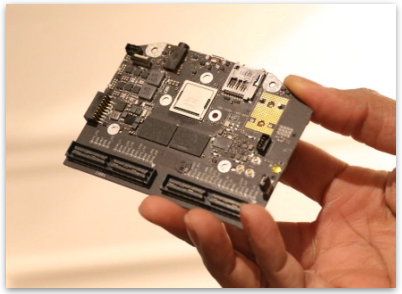
- Can be desirable to scale problem size as machine sizes grow

  (buy a bigger machine to compute more, rather than just compute the same problem faster)

Architects also think about scaling

A common question: "Does an architecture scale?"

- **Scaling up:** how does architecture's performance scale with increasing core count?
  - Will design scale to the high end?
- **Scaling down:** how does architecture's performance scale with decreasing core 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
    - Same SMM core architecture, different numbers of SMM cores per chip



Tegra X1: 2 SMM cores
(mobile SoC)

GTX 950: 6 SMM cores
(90 watts)

GTX 980: 16 SMM cores
(165 watts)

Titan X: 24 SMM cores
(250 watts)

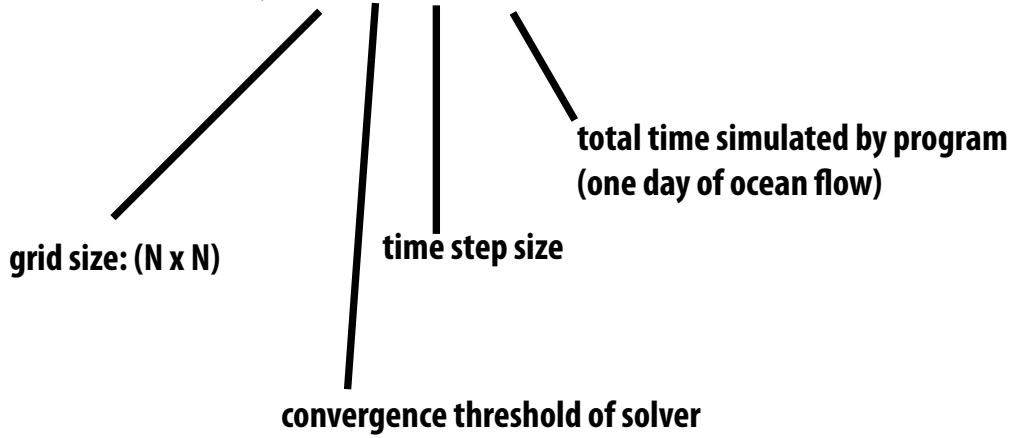
Common Scaling Terminology

- For mapping problem X on system with P processors
- Strong scaling
  - Runtime of X on P processors, vs. of X on 1 processor
    - Goal = P
  - Does having more processors get job done faster?
- Weak scaling
  - Runtime of (P*X) on P processors, vs. of X on 1 processor
    - Goal = 1.0
  - Does having more processors let me do bigger jobs?

Questions to ask when scaling a problem

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

- **How should be the problem be scaled?**
  - Problem size is often determined by more than one parameter
  - Ocean example: problem defined by (N, ε, Δt, T)



Scaling constraints

- Application-oriented scaling properties (specific to application)
  - 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, and apply across domains. (so we'll talk about them today)

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 may not be realistic workloads for large machines, big problems may not fit on small machines)

- Examples of problem-constrained scaling:
  - Almost everything we've considered parallelizing in class so far
  - Assignments 1 and 2 (and 3)

Time-constrained scaling

- Focus: completing 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"?
  - Challenge: "work done" may not be linear function of values of problem inputs (e.g. matrix multiplication is $O(N^3)$ work for $O(N^2)$ sized inputs)
  - One approach: "work done" is defined by execution time of same computation on a single processor (but consider effects of thrashing if problem too big)
  - Ideally, a measure of work is:
    - Simple to understand
    - Scales linearly with sequential run time (So ideal speedup remains linear in P)

Time-constrained scaling example

Real-time 3D graphics: more compute power allows for rendering of much more complex scene

Problem size metrics: number of polygons, texels sampled, shader length, etc.

Half-Life (1998)

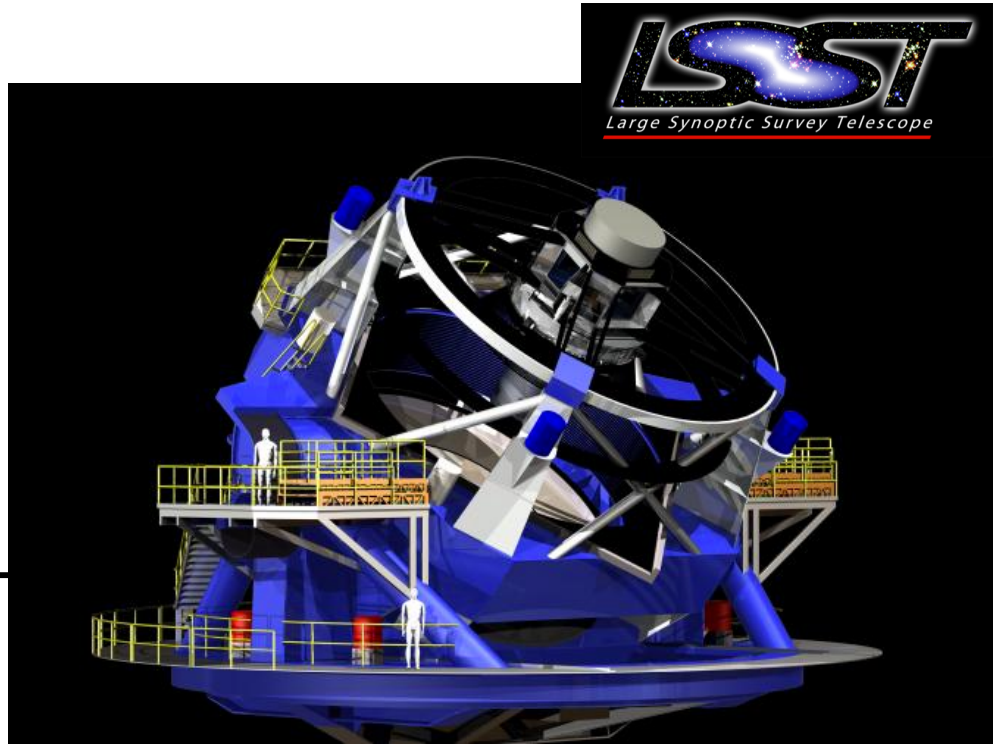
Assassin's Creed Unity (2014)

Image credits:
http://www.gamespot.com/forums/system-wars-314159282/assassin-s-creed-unity-best-graphics-of-2014-31696528/
http://www.game-weavers.com/?page_id=490

Time-constrained scaling example

Large Synoptic Survey Telescope (LSST)

- Estimated completion in 2019
- Acquire high-resolution survey of sky (3-gigapixel image every 15 seconds, every night for many years)



LSST will be located on top of Cerro Pachón Mountain, Chile

Rapid Image analysis compute platform (detect "potentially" interesting events)

Notify other observatories if potential event detected.

Increasing compute capability allows for more sophisticated detection algorithms (fewer false positives, detect broader class of events)

Image credits:
http://www.lsst.org
http://mcdonaldobservatory.org

More time-constrained scaling examples

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

- **Modern web sites**
  - Want to generate complex page, respond to user in X milliseconds (studies show site usage directly corresponds to page load latency)

- **Real-time computer vision for robotics**
  - Consider self-driving car: best quality pedestrian detection in 10 ms

Memory-constrained scaling

- Focus: run the largest problem possible without overflowing main memory **
- Memory per processor is held fixed (e.g., add more machines to cluster)
- *Neither work nor execution time are held constant!*

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

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

- 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)

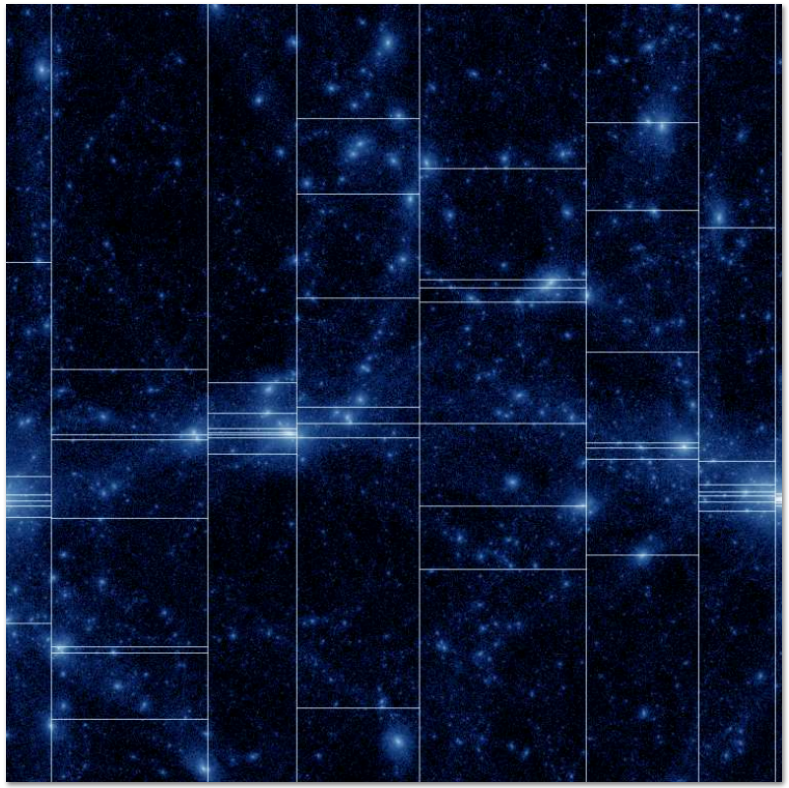
Memory-constrained scaling examples

- One motivation to use supercomputers and large clusters is simply to be able to fit large problems in memory

- Large N-body problems
  - 2012 Supercomputing Gordon Bell Prize Winner: 1,073,741,824,000 particle N-body simulation on K-Computer (MPI implementation)

- Large-scale machine learning
  - Billions of clicks, documents, etc.

- Memcached (in memory caching system for web apps)
  - More servers = more available cache



2D domain decomposition of N-body simulation

Image credit: Ishiyama et al. 2012

Scaling examples at PIXAR

- **Rendering a "shot" (a sequence of frames) in a movie**
  - Goal: minimize 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 to artist (time constrained)
  - More performance = higher fidelity representation shown to artist in allotted time

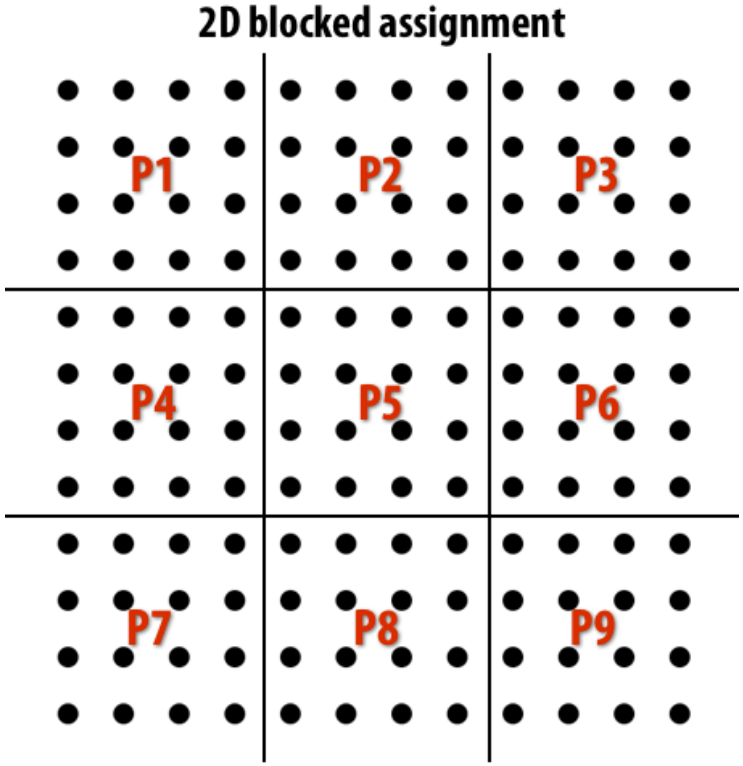
- **Physical simulation: e.g., fluid simulation**
  - Parallelize simulation across multiple machines to fit simulation grid in aggregate memory of processors (memory constrained)

- **Final render of images for movie**
  - Scene complexity is typically bounded by memory available on farm machines
  - One barrier to exploiting additional parallelism within a machine is that required footprint often increases with number of processors (consider your implementation of assignment 2!)

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
  - Elements per processor: N^2/P
  - Available 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)

Scaling the 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**: (note: N is constant here)
  - Execution time: O(1/P)
  - Elements per processor: O(1/P)
  - Communication per processor: $O(1/P^{1/2})$
  - Comm-to-comp ratio: $O(P^{1/2})$
    (for a 2D-blocked assignment)

notice: execution time increases with MC scaling

- **Memory-constrained scaling**:
  - Let scaled grid size be $NP^{1/2}$ x $NP^{1/2}$
  - Execution time: $O((NP^{1/2})^3/P) = O(P^{1/2})$
  - Elements per processor: fixed! (N^2)
  - Comm-to-comp ratio: O(1)

- **Time-constrained scaling**:
  - Let scaled grid size be K x K
  - Assume linear speedup: $K^3/P = N^3$ (so $K = NP^{1/3}$)
    (recall 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)
  - Elements per processor: $K^2/P = N^2/P^{1/3}$
  - Communication per processor: $K/P^{1/2} = O(1/P^{1/6})$
  - 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
  - Recall Ocean example: problem is $=(n, \varepsilon, \Delta t, T)$
- 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 under situations of 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 the scaling properties of the problem and also the machine.
  - When analyzing performance, be careful to analyze realistic regimes of operation (realistic sizes and realistic problem size parameters)
  - This 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

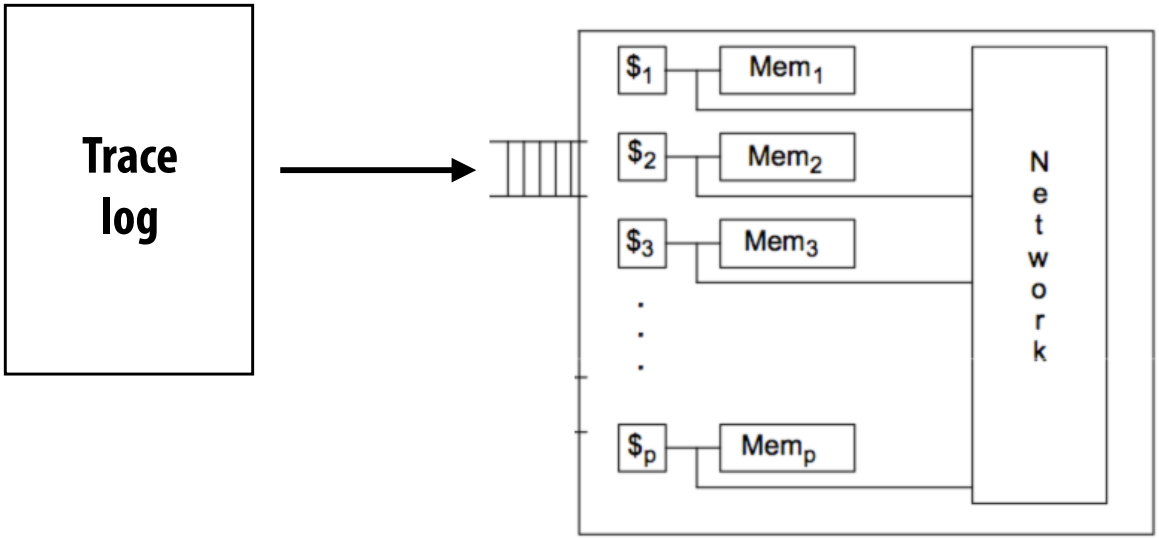
- Architects evaluate architectural decisions quantitatively using hardware performance simulators
  - Let's say an architect is considering adding a feature
  - Architect runs simulations with new feature, runs simulations without new feature, compare simulated performance
  - Simulate against a wide 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?)

Trace-driven simulator

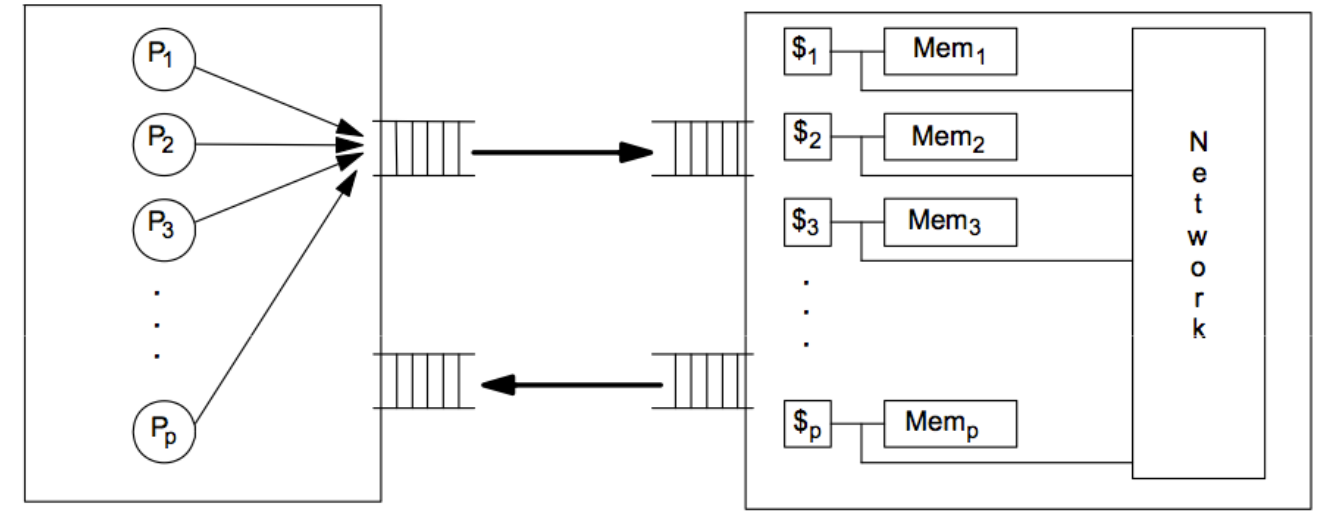
- Instrument real code running on real machine to record a trace of all memory accesses
  - Statically (or dynamically) modify program binary
  - Example: Intel's PIN (www.pintool.org)
  - May also need to record timing information to model contention in subsequent simulation

- Then play back trace on simulator



Drawbacks?

Execution-driven simulator

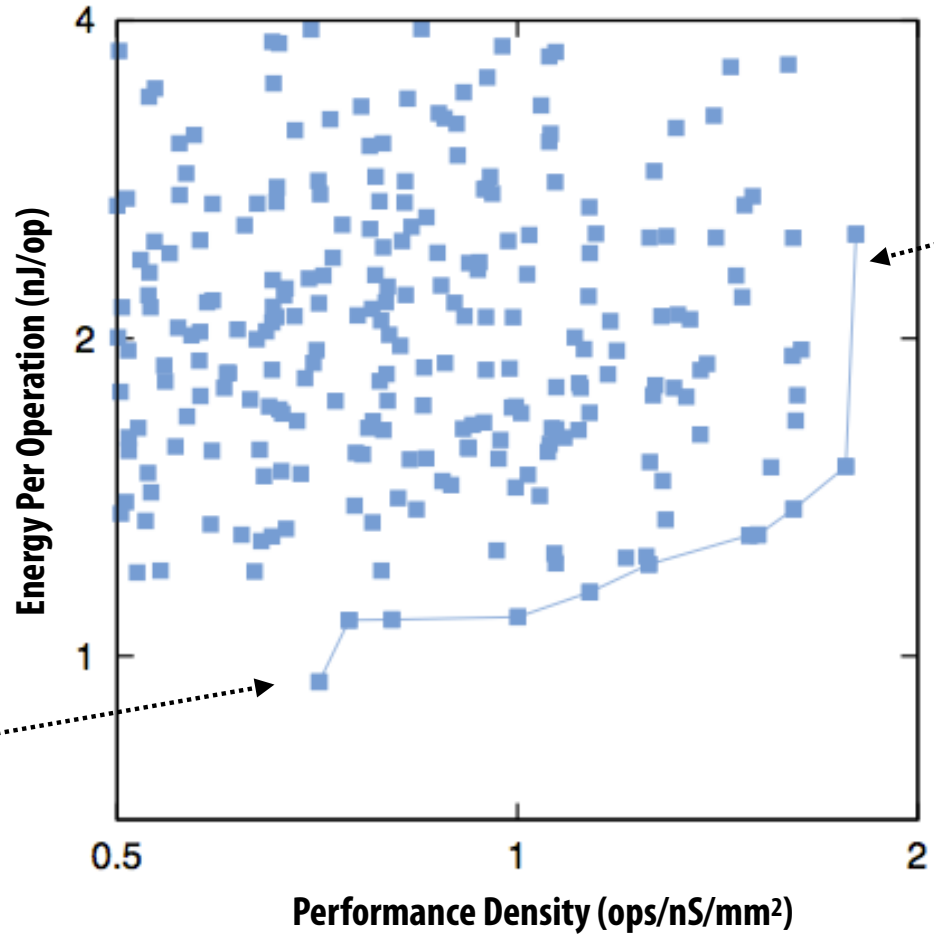


- Executes simulated program in software
  - Simulated processors generate memory references, which are processed by the simulated memory hierarchy
- Performance of simulator is typically inversely proportional to level of simulated detail
  - Simulate every instruction? Simulate every bit on every wire?

Architectural simulation state space

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

Pareto Curve: (here: plots energy/perf trade-off)



Maximum performance per unit chip area (but high energy consumption)

High energy efficiency (but lower performance density)

The challenges of scaling problems up or down also apply to software developers debugging/ tuning parallel programs on real machines

Common examples:

- **May want to log behavior of code when debugging**
  - **Debug logs get untenable in size when running full problem**
  - **Instrumentation slows down code significantly**
  - **Instrumentation removes contention by changing timing of application**

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

Challenges of scaling down (or up)

- **Preserve ratio of time spent in different program phases**
  - Assignment 2: "bin circles" phase, process lists phase
  - 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

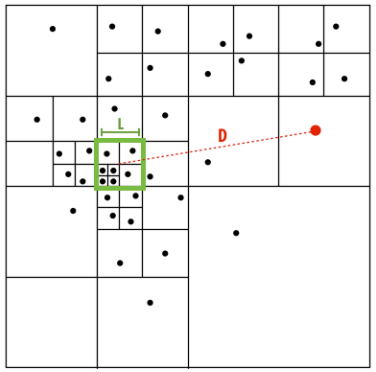
- **Preserve important behavioral characteristics**
  - arithmetic intensity, load balance, locality, working set sizes
  - e.g., shrinking grid size in solver changes arithmetic intensity

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

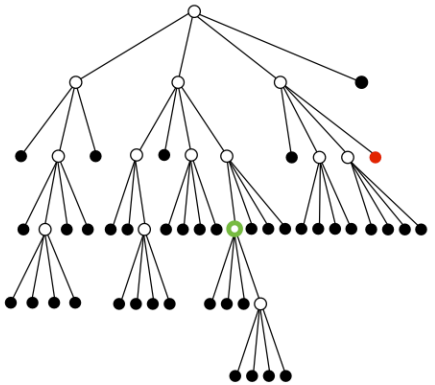
- **Preserve scaling relationships between problem parameters**
  - e.g., Barnes-Hut: scaling up particle count requires scaling down time step for physics reasons

Example: scaling down Barnes-Hut

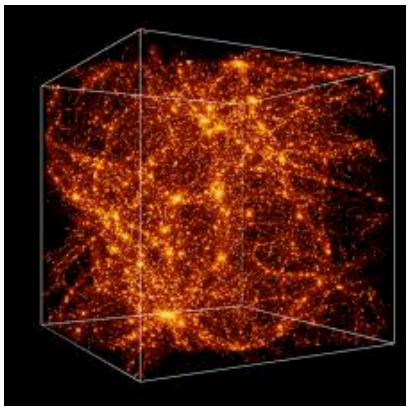
- **Problem size = (n, Θ, Δt, T)**
  - n: grid size
  - Θ: accuracy threshold
  - Δt: time step
  - T: total time to simulate



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 brings particles together over time and performance depends on particle density)
  - <u>One solution</u>: select a few representative periods of time (e.g., sequence of time steps at beginning of sim, at end of sim)

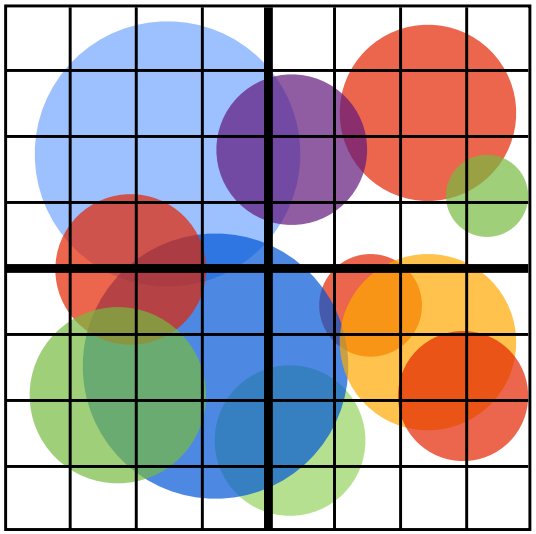


Example: scaling down assignment 2

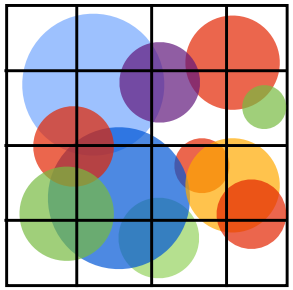
- **Problem size = (w, h, num_circles, ...)**

image width, height

number of scene circles

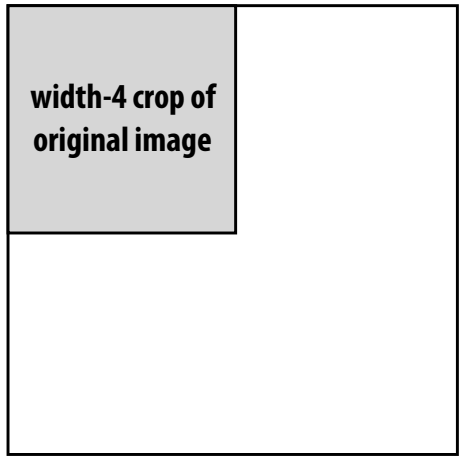


Original image
(w=8)



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

Alternative solution:
Render crop of full-size image



Original image
(w=8)

Shrink image, but keep circle data the same:

- Ratio of circle/box "filtering" work to per-pixel work changes
- With fixed tile size: number of circles in a tile changes

There is simply no substitute for the experience of writing and tuning your own parallel programs. But today's take-away is: BE CAREFUL!

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

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

It is helpful to start by precisely stating your application performance goals. Then determine if your evaluation approach is consistent with these goals.

Here are some tricks for understanding the performance of parallel software

Always, always, always try the simplest parallel solution first, then measure performance to see where you stand.

A useful performance analysis strategy

- 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 in practice?
  - How close is your implementation to a best-case scenario?

Roofline model

- Use microbenchmarks to compute peak performance of a machine as a function of arithmetic intensity of application
- Then compare application's performance to known peak values

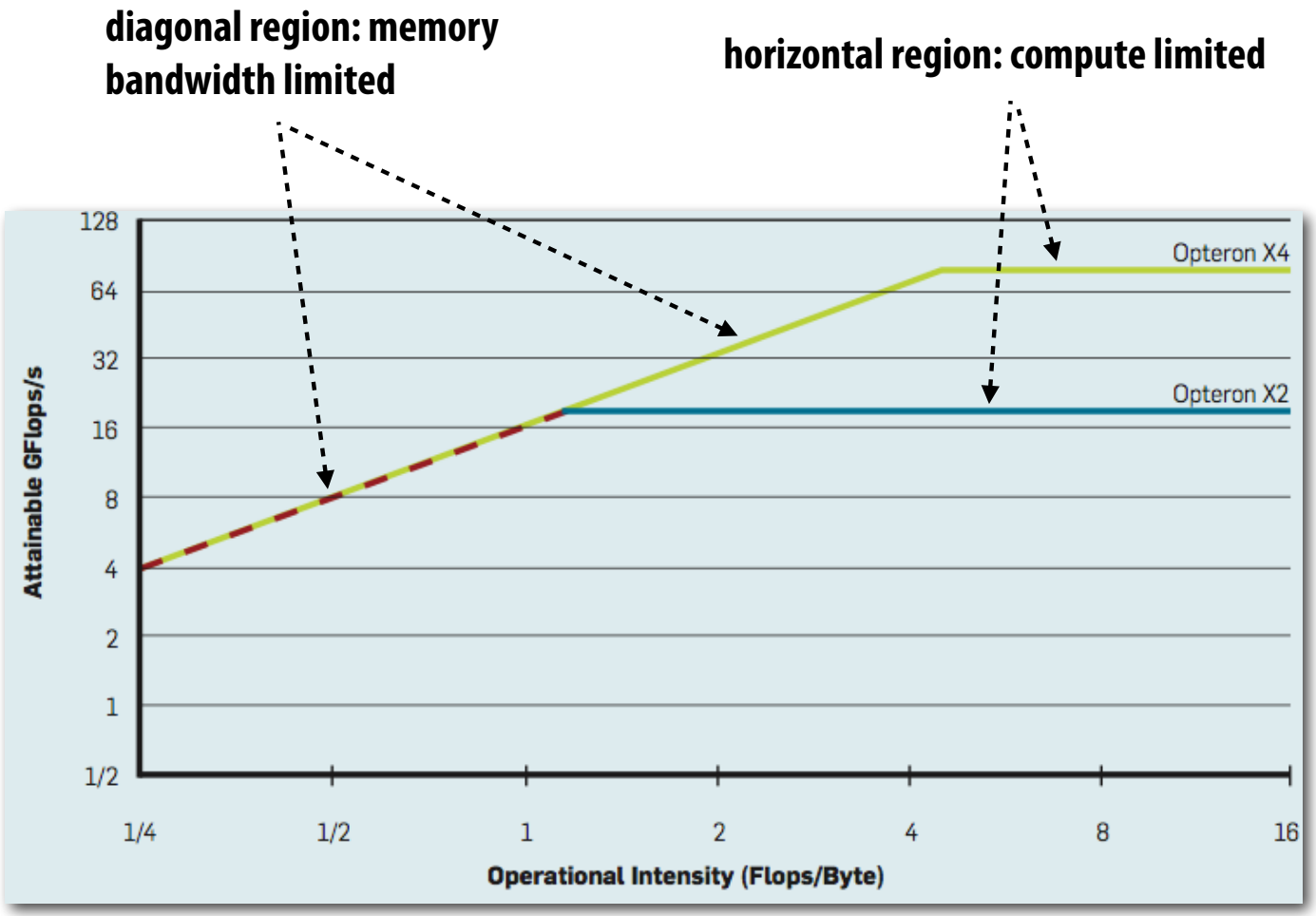


Figure credit: Williams et al. 2009

Roofline model: optimization regions

- Use various levels of optimization in benchmarks
  (e.g., best performance with and without using SIMD instructions)

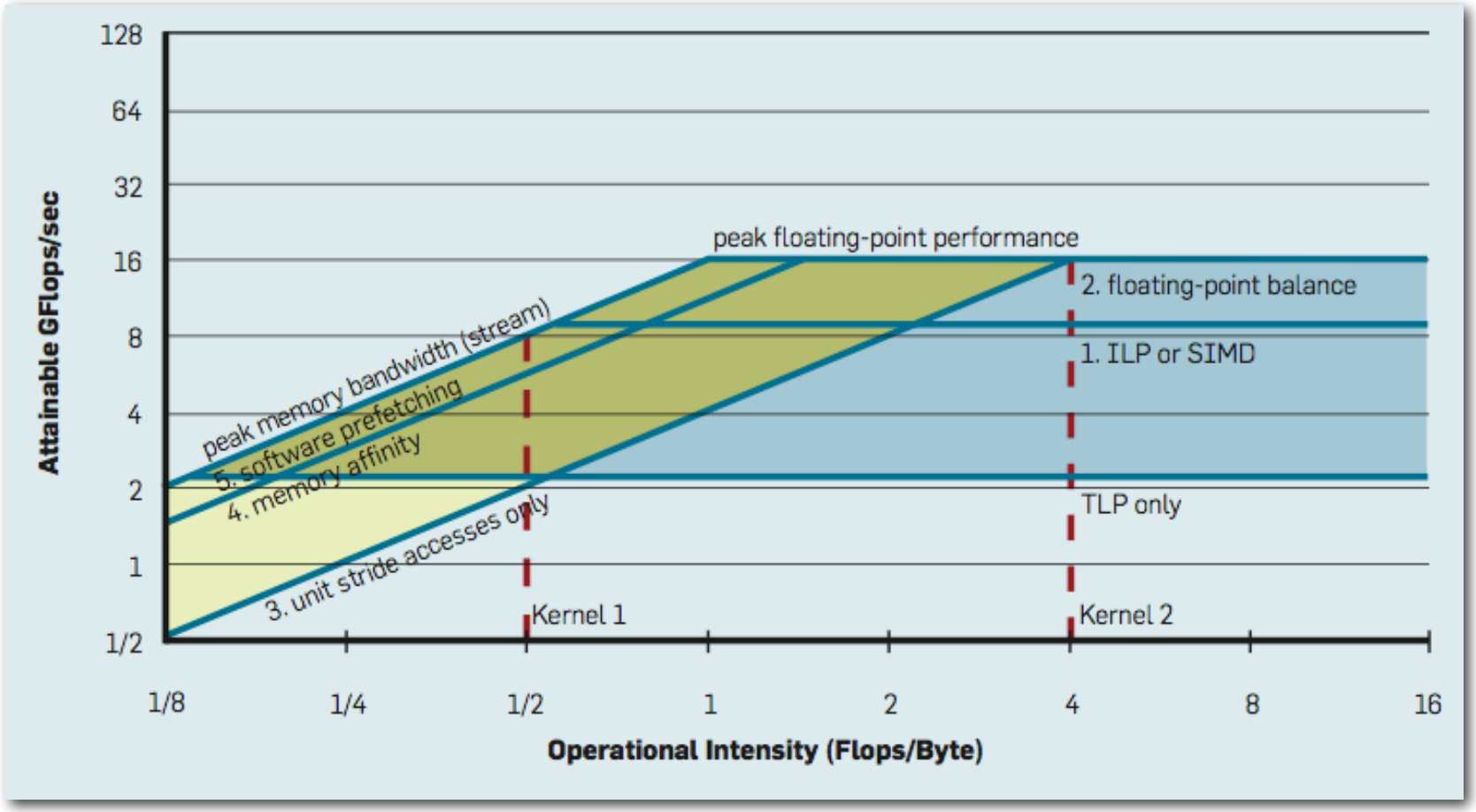


Figure credit: Williams et al. 2009

Establishing high watermarks *

Add "math" (non-memory instructions)

Does execution time increase linearly with operation count as math is added?
(If so, this is evidence that code is instruction-rate limited)

Remove almost all math, but load same data

How much does execution-time 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 improving locality of data access)

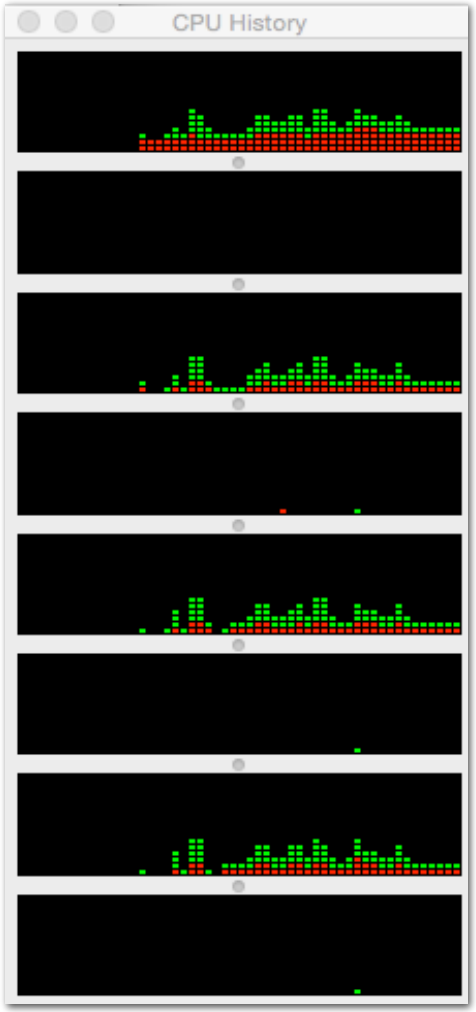
Remove all atomic operations or locks

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.)

* Computation, memory access, and synchronization are almost never perfectly overlapped. As a result, overall performance will rarely be dictated entirely by compute or by bandwidth or by sync. Even so, the sensitivity of performance change to the above program modifications can be a good indication of dominant costs

Use profilers/performance monitoring tools

- Image at left is "CPU usage" from activity monitor in OS X while browsing the web in Chrome (laptop has a quad-core Core i7 CPU)
  - Graph plots percentage of time OS has scheduled a process thread onto a processor execution context
  - Not very helpful for optimizing performance



- All modern processors have low-level event "performance counters"
  - Registers that count important details such as: instructions completed, clock ticks, L2/L3 cache hits/misses, bytes read from memory controller, etc.

- Example: Intel's Performance Counter Monitor Tool provides a C++ API for accessing these registers.

```
PCM *m = PCM::getInstance();
SystemCounterState begin = getSystemCounterState();

// code to analyze goes here

SystemCounterState end = getSystemCounterState();

printf("Instructions per clock: %f\n", getIPC(begin, end));
printf("L3 cache hit ratio: %f\n", getL3CacheHitRatio(begin, end));
printf("Bytes read: %d\n", getBytesReadFromMC(begin, end));
```

- Also see Intel VTune, PAPI, oprofile, etc.