

Lecture 14:

Workload-Driven Performance Evaluation

**Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2026**

Reminders

- **5-minute meetings with TAs to discuss assignment 2 design**

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?**

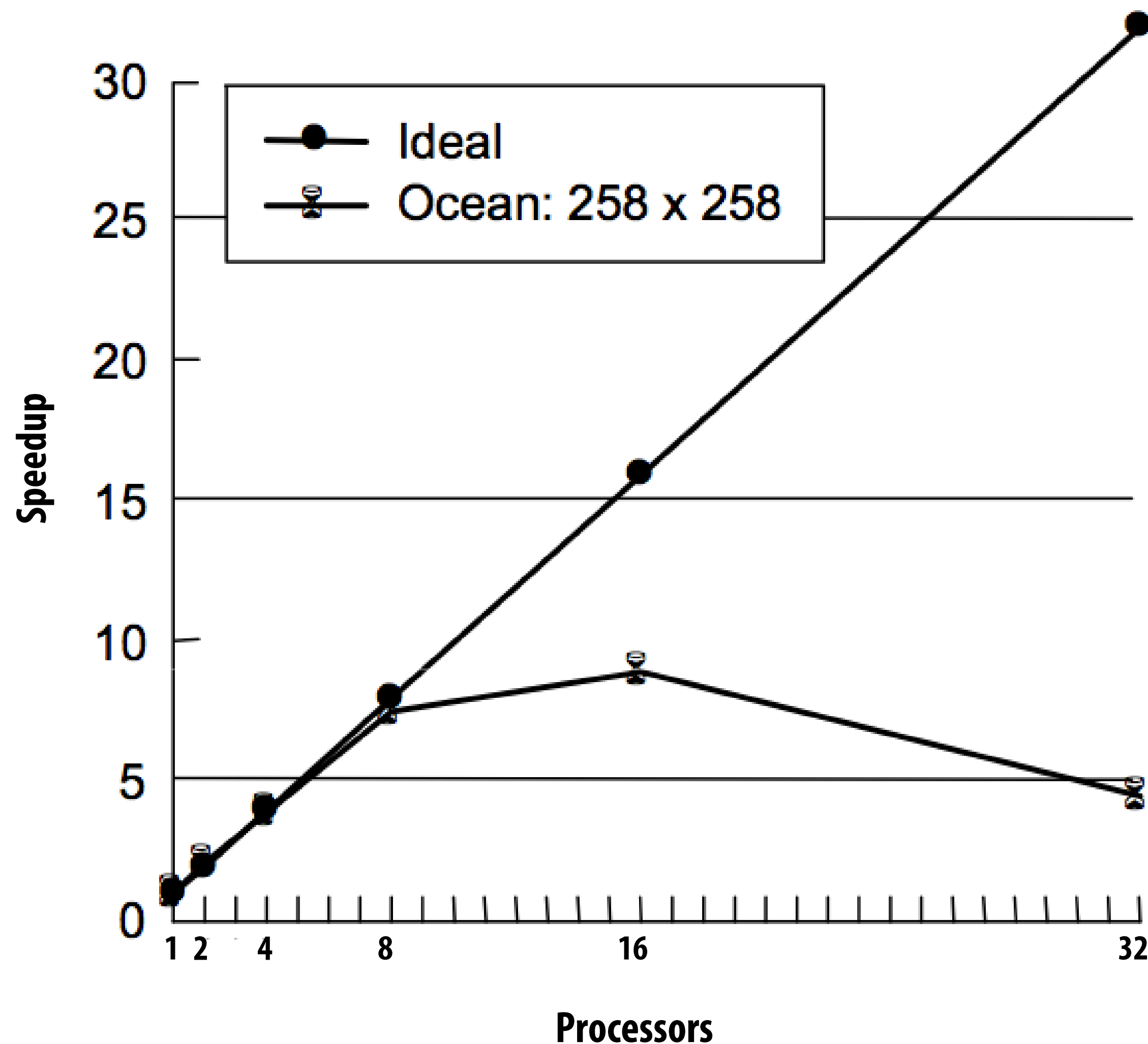
- $\text{Execution time of sequential program} / \text{execution time on } P \text{ processors}$
- $\text{Operations per second on } P \text{ processors} / \text{operations per second of sequential program}$

- **Efficiency?**

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

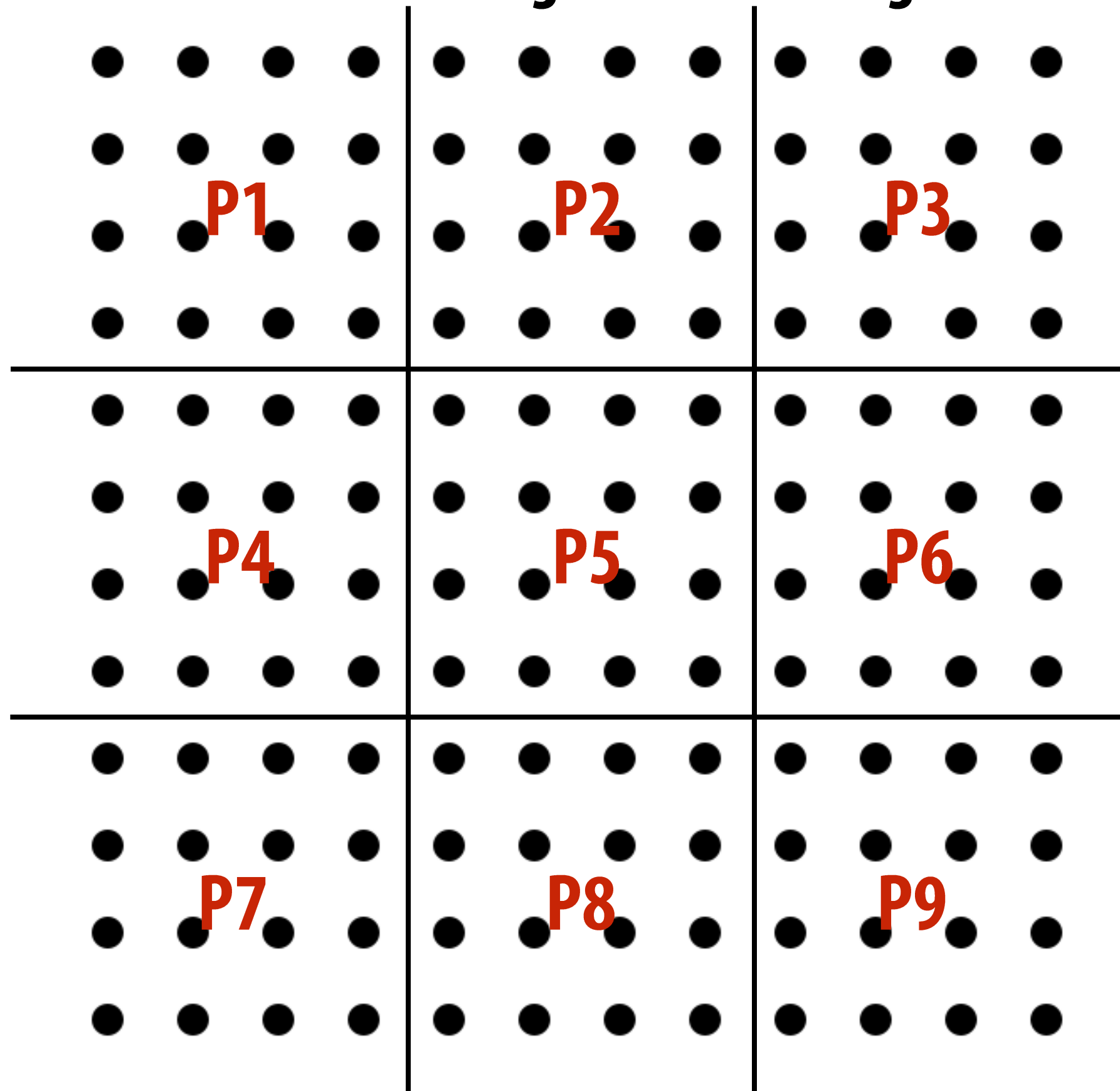
Speedup of ocean sim application: 258 x 258 grid

Execution on 32 processor SGI Origin 2000



Remember: work assignment in ocean

2D blocked assignment: $N \times N$ grid



N^2 elements

P processors

elements computed:
(per processor)

$$\frac{N^2}{P}$$

elements communicated:
(per processor)

$$\propto \frac{N}{\sqrt{P}}$$

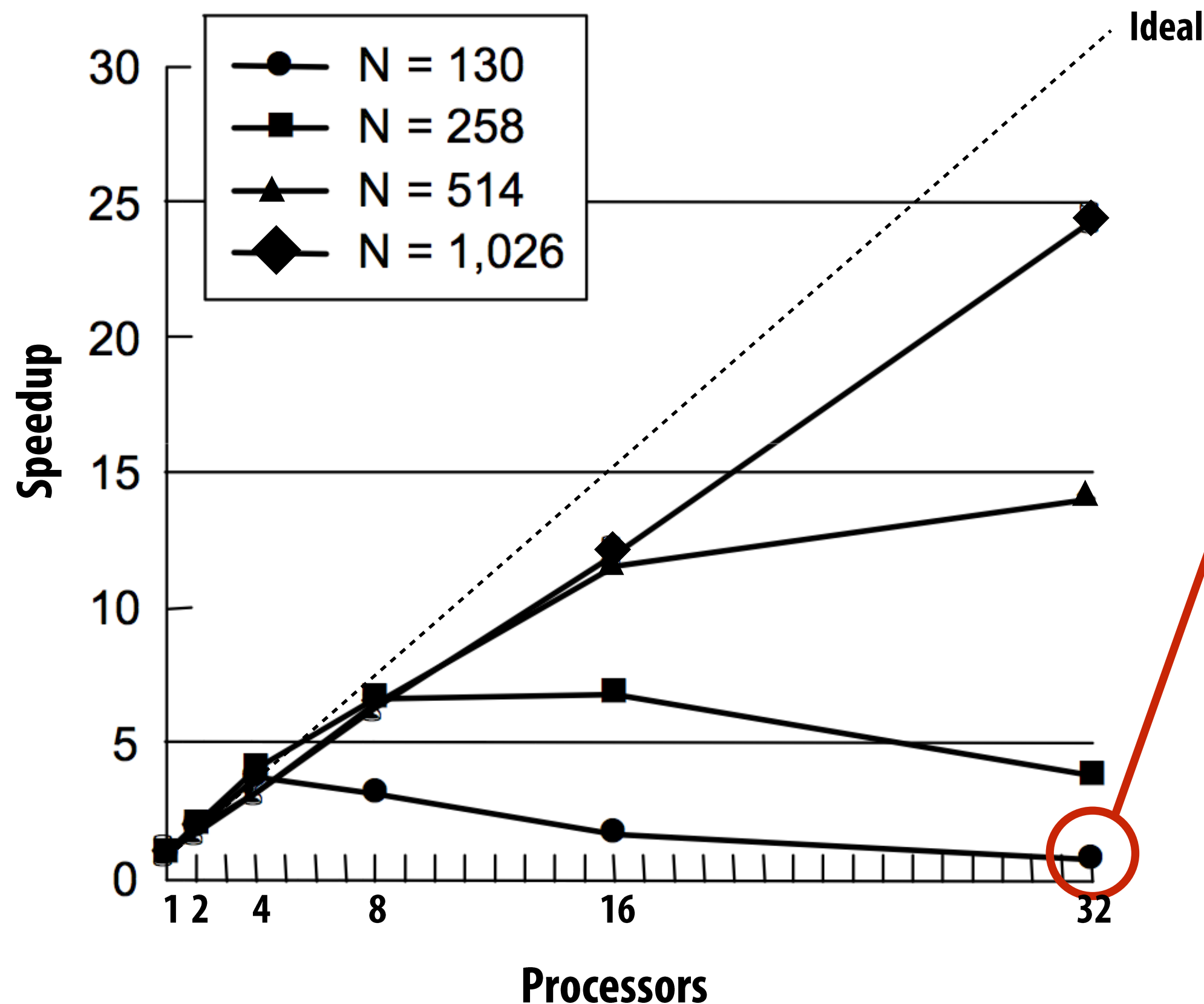
arithmetic intensity:

$$\frac{N}{\sqrt{P}}$$

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)**

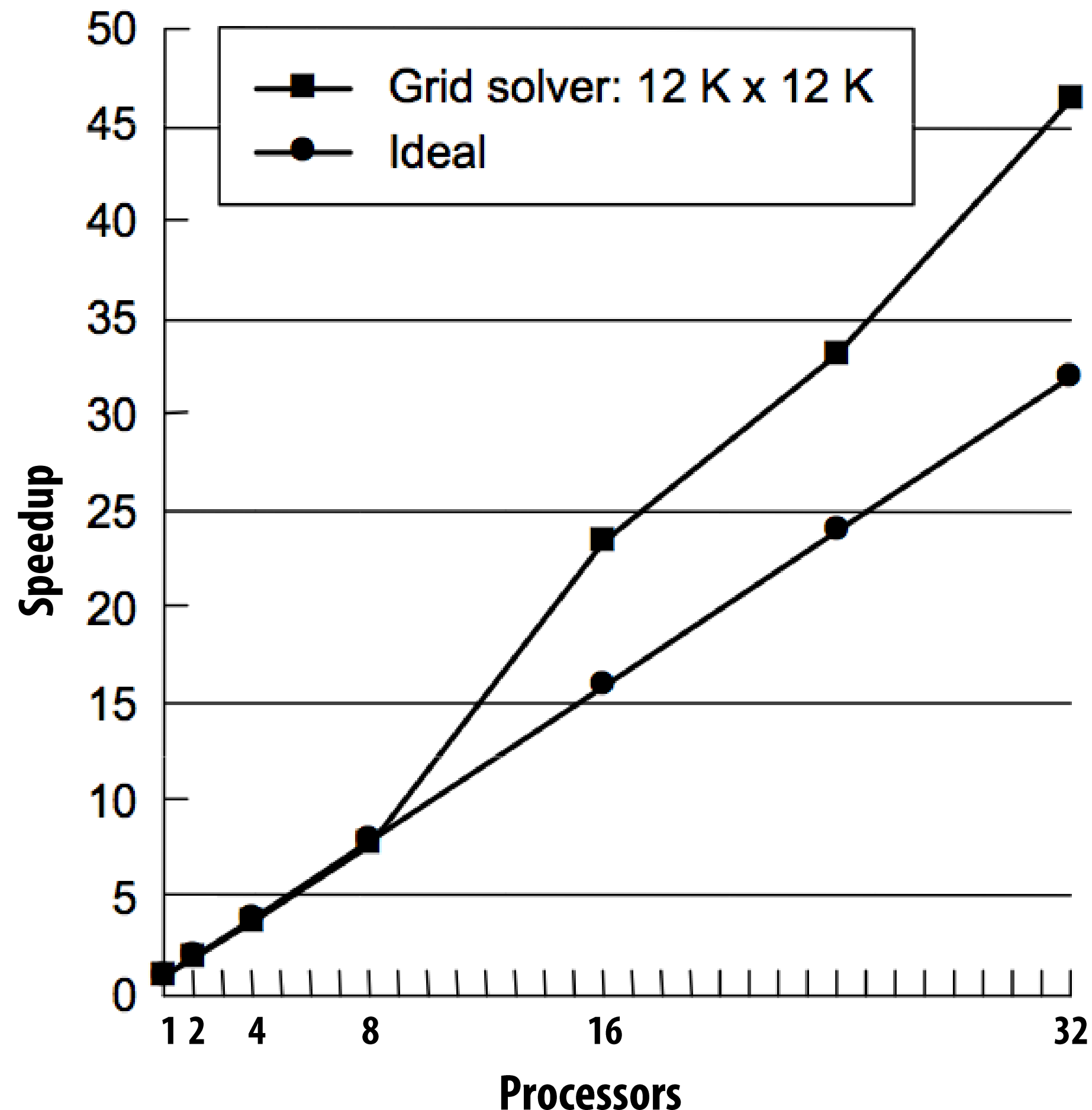
**Scaling the performance of small problem may
not be all that important anyway (it might
already execute fast enough on a single core)**

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

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

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!)

Understanding scaling: size matters!

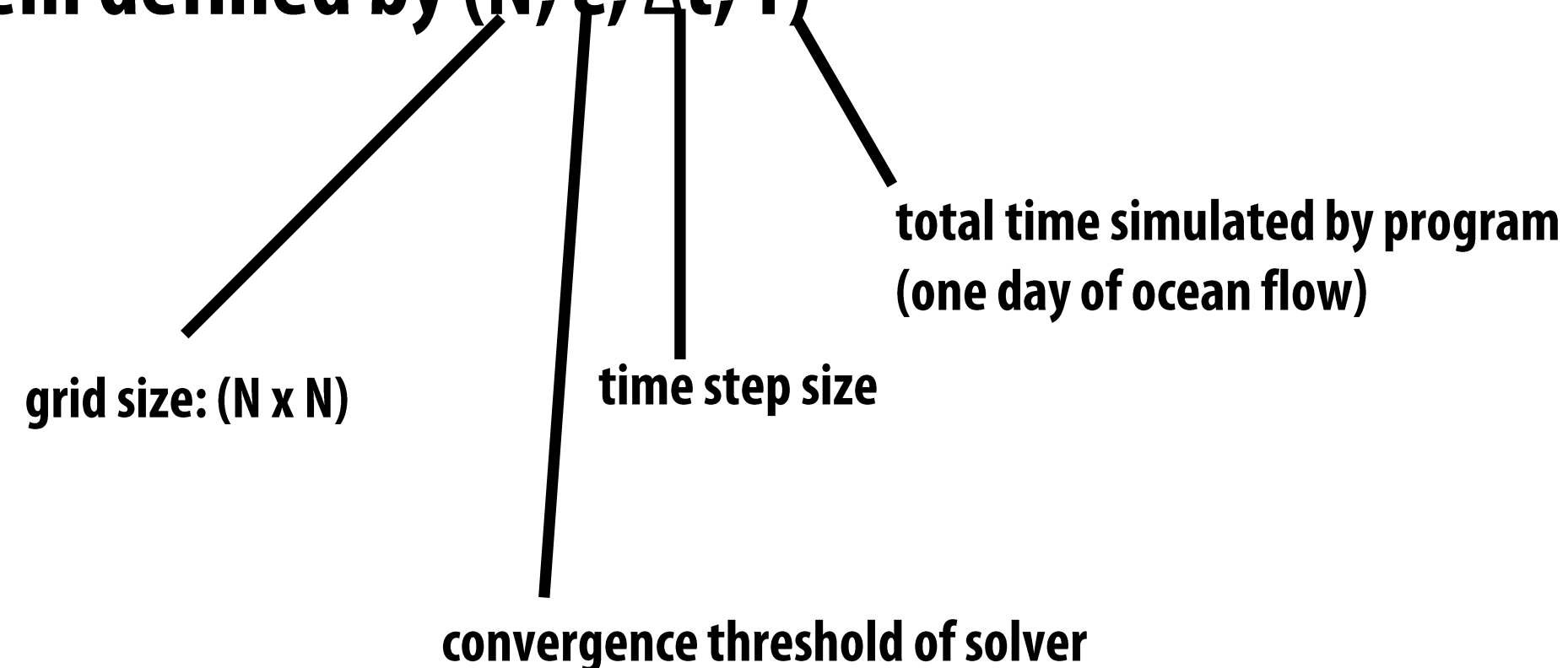
- There can be **complex interactions between the size of the problem and the size of the parallel computer**.
 - Can impact load balance, overhead, arithmetic intensity, locality of data access
 - Effects can be dramatic and application dependent
- **Evaluating a machine with a fixed problem size can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

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 = $P\times$ faster, or $1/P\times$ runtime
 - *Q. Does having more processors get job done faster?*
- **Weak** scaling
 - Runtime of $(P\times X)$ on P processors, vs. of X on 1 processor
 - Goal = $1.0\times$ runtime
 - *Q. 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 the problem be scaled?**
 - Problem size is often determined by more than one parameter
 - Ocean example: problem defined by $(N, \epsilon, \Delta t, T)$



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

* Problem-constrained scaling is often called “hard scaling”.

Time-constrained scaling *

- **Focus: completing more work in a fixed amount of time**

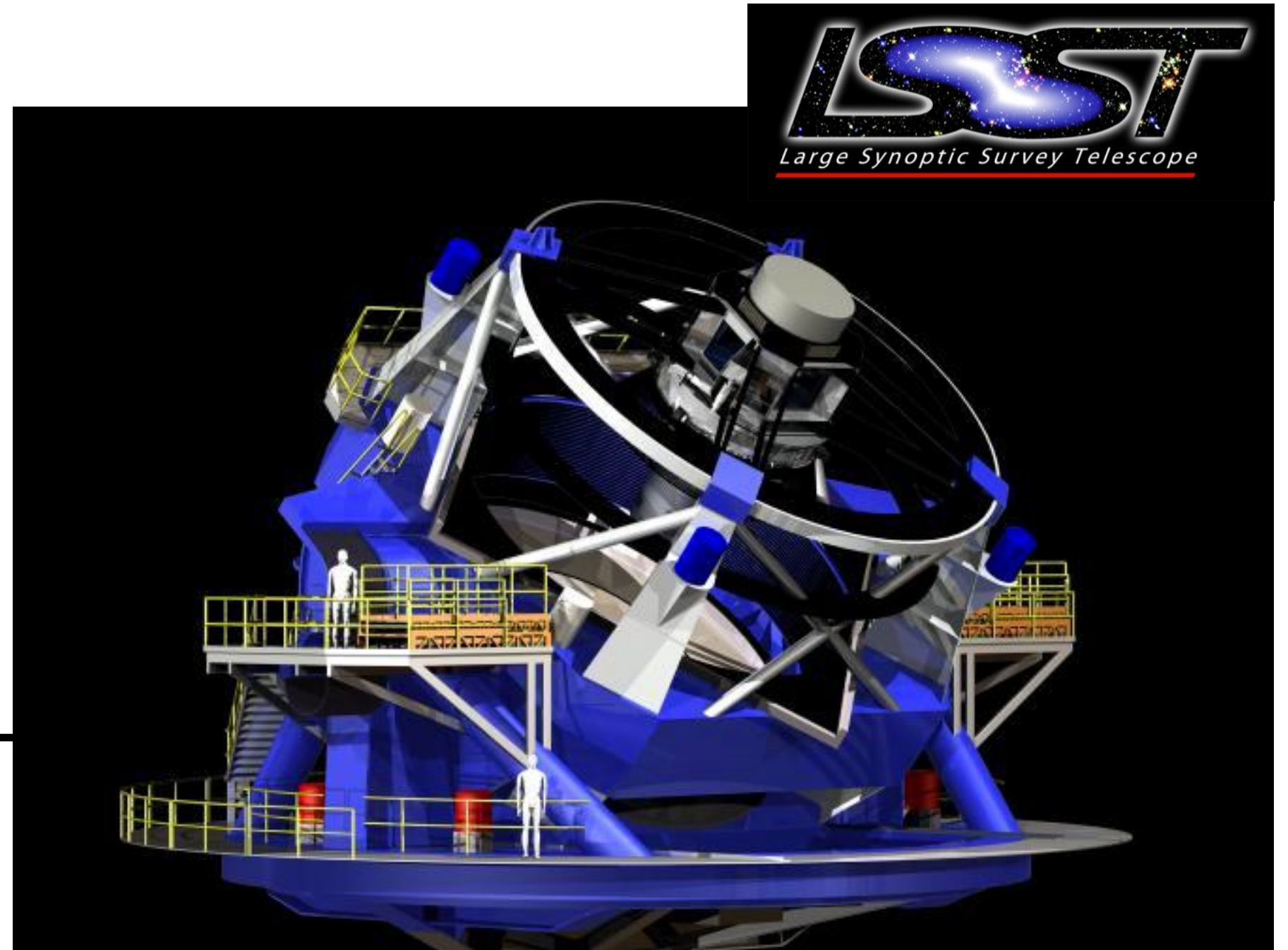
$$\text{Speedup} = \frac{\text{work done by } P \text{ processors}}{\text{work done by 1 processor}}$$

- Execution time kept fixed as the machine (and problem) scales
- **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

Large Synoptic Survey Telescope (LSST)

- Came online in April 2025
- 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)

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)

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

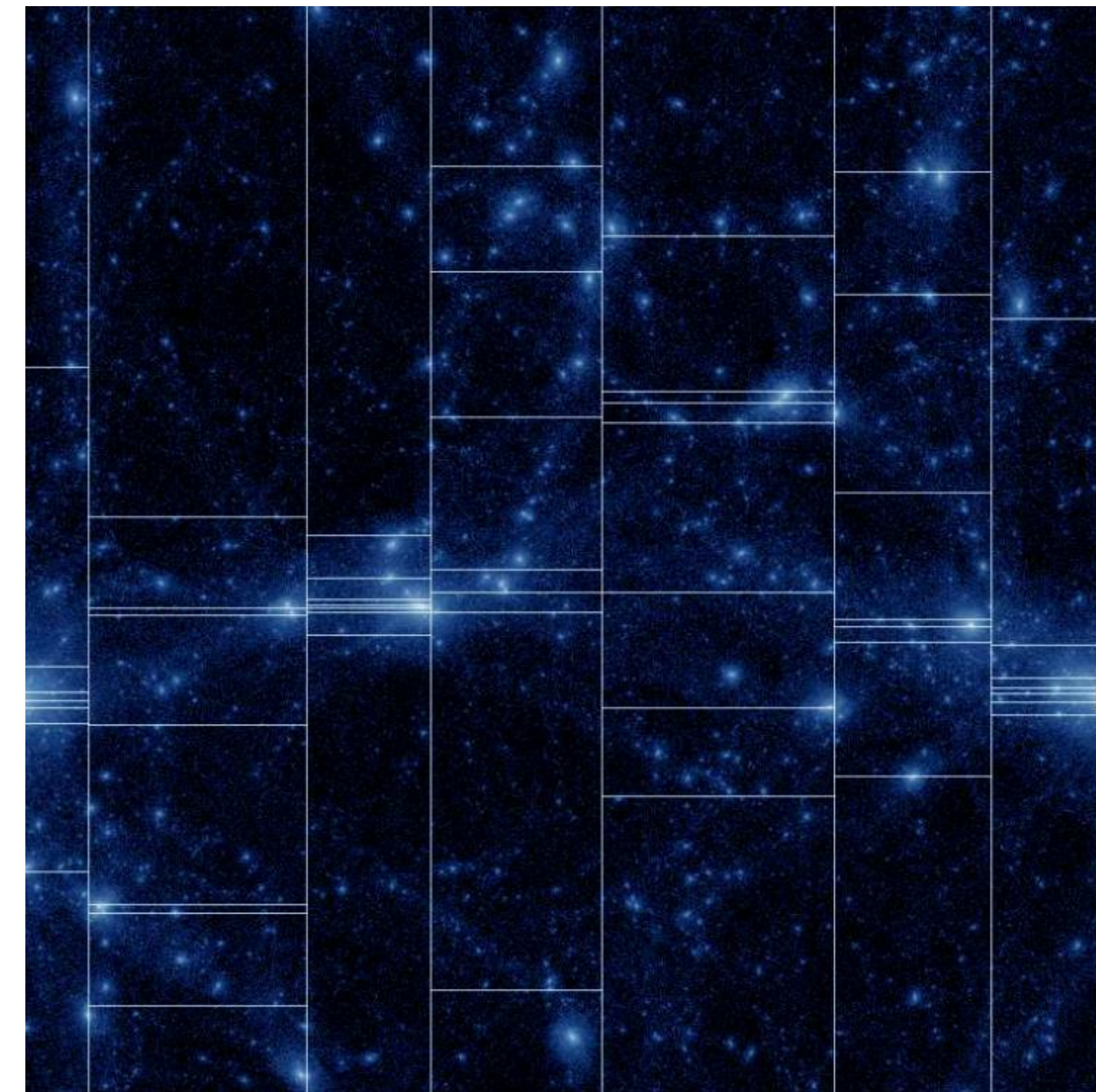
- Neither work or execution time are held constant
- Note: scaling problem size can make runtimes very large
 - Consider $O(N^3)$ matrix multiplication on $O(N^2)$ matrices

* Memory-constrained scaling is often called “weak scaling”

** 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

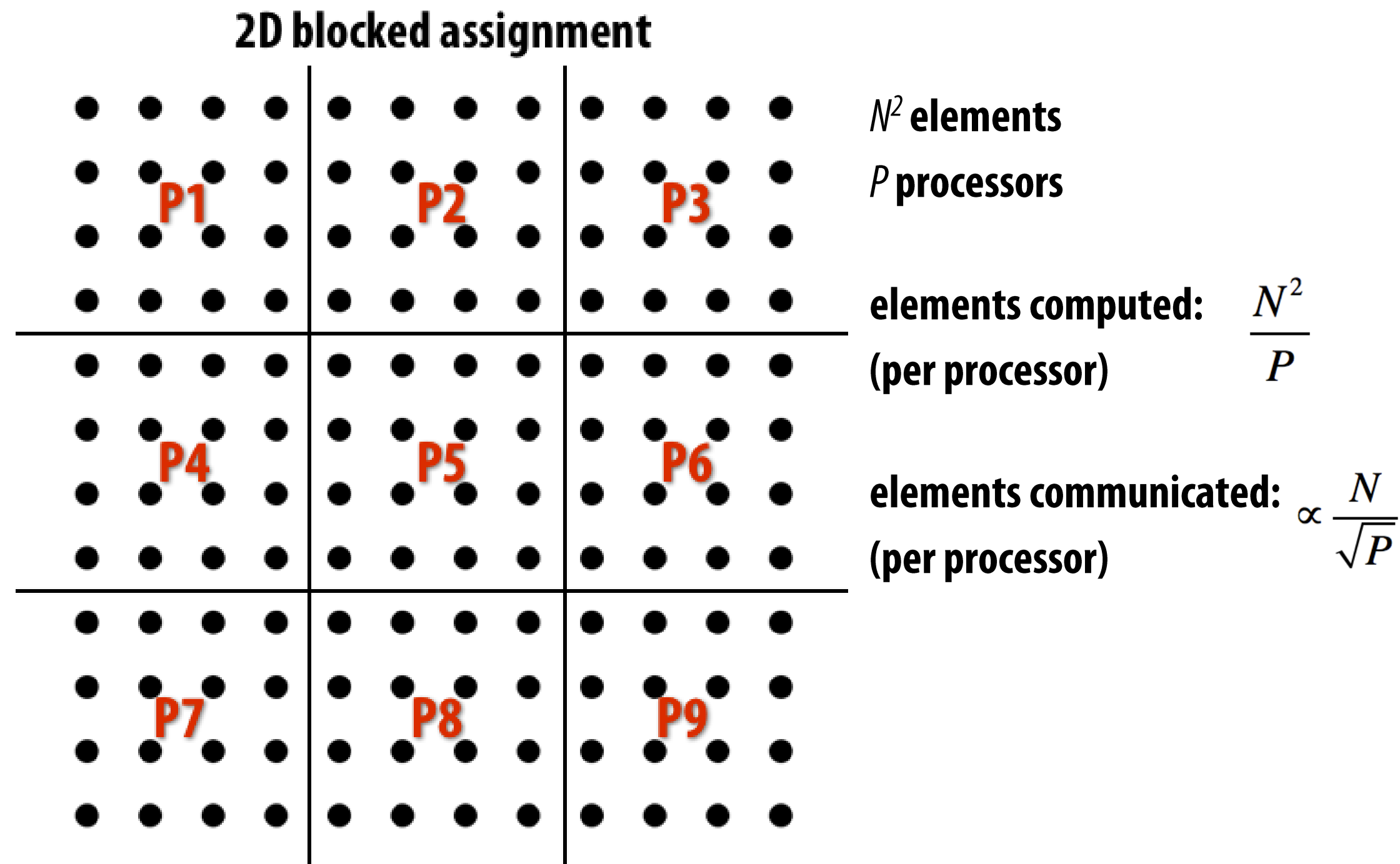


2D domain decomposition of N-body simulation

Case study: our 2D grid solver

- For $N \times N$ 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)



Scaling the 2D grid solver

- For $N \times N$ grid:
 - $O(N^2)$ memory requirement
 - $O(N^2)$ grid elements $\times$ $O(N)$ 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)
 - **Memory-constrained scaling:**
 - Let scaled grid size be $NP^{1/2} \times NP^{1/2}$
 - Execution time: $O((NP^{1/2})^3/P) = O(P^{1/2})$
 - Elements per processor: fixed! (N^2)
(by definition of memory-constrained scaling)
 - **Comm-to-comp ratio: fixed at $1/N$**
 - **Time-constrained scaling:**
 - Let scaled grid size be $K \times K$
 - Assume linear speedup: $K^3/P = N^3$ (so $K = NP^{1/3}$)
(recall computation time for $K \times K$ grid on P processors = computation time for $N \times N$ 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: $O(P^{1/6})$**
- notice: execution time increases with MC scaling
- 

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 $=(n, \epsilon, \Delta t, T)$
- 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

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

Common examples:

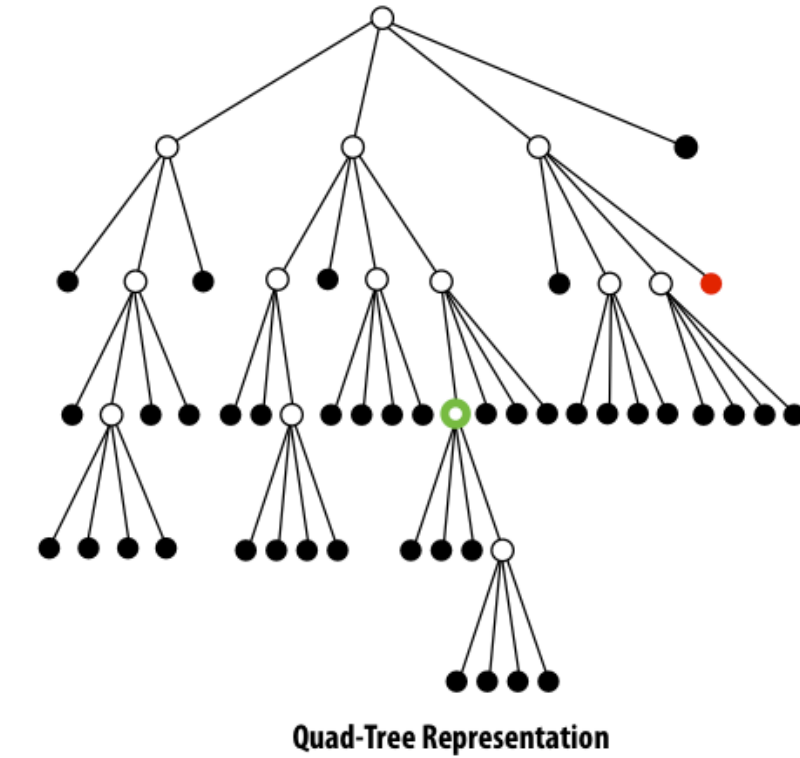
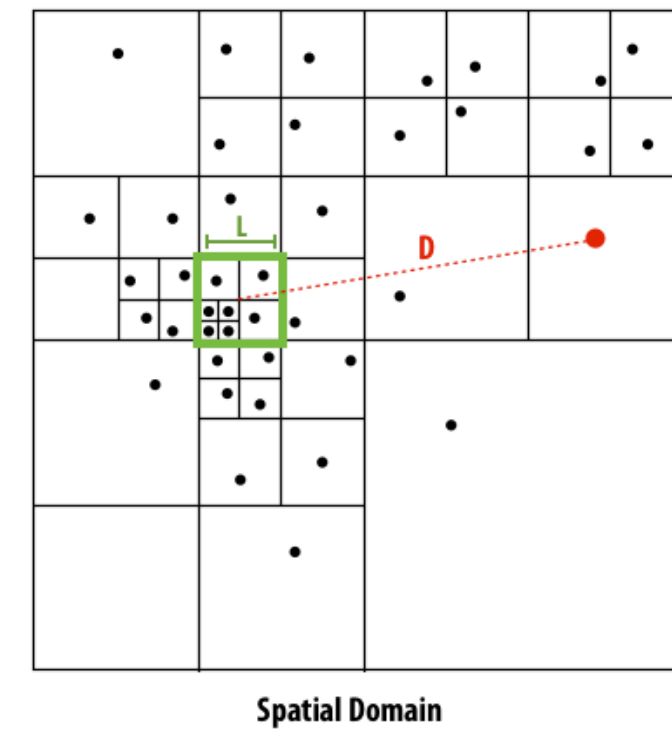
- **May want to log behavior of code when debugging**
 - **Debug logs get untenable in size when running full problem**
 - **Instrumentation slows down code significantly**
 - **Instrumentation removes contention by changing timing of application**
- **May want to debug/test/tune code on small problem size on small machine before running two-week simulation on full problem size on a supercomputer**

Challenges of scaling down (or up)

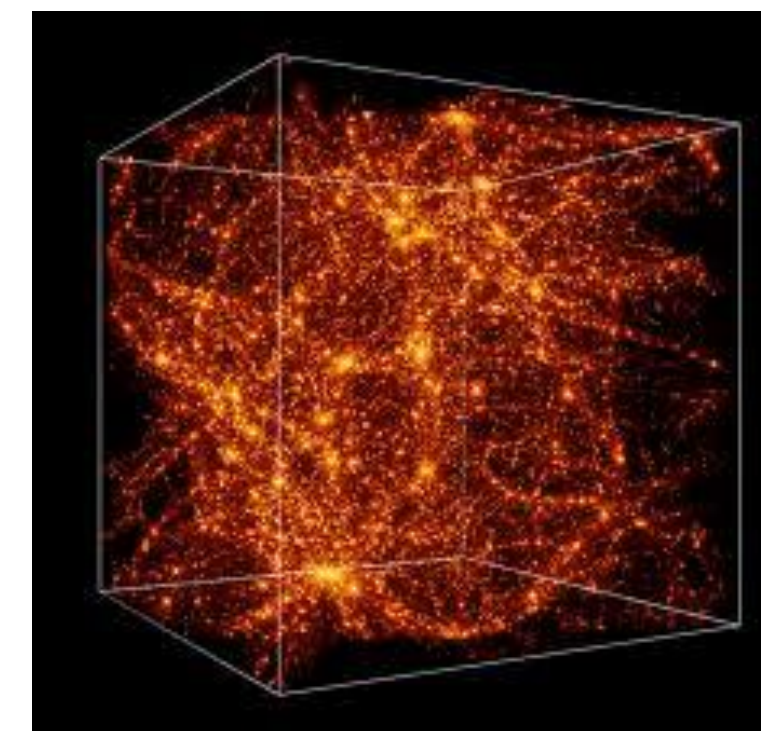
- **Preserve ratio of time spent in different program phases**
 - Assignment 2: “bin circles” phase, process lists phase
 - Ray-trace and Barnes-Hut: both have tree build and tree traverse phases
 - Shrinking screen size changes cost of tracing rays but not cost of building tree
 - Changing density of 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 = $(n, \epsilon, \Delta t, T)$**
 - grid size
 - accuracy threshold
 - time step
 - total time to simulate



- **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)
 - One solution: select a few representative periods of time (e.g., sequence of time steps at beginning of sim, at end of sim)

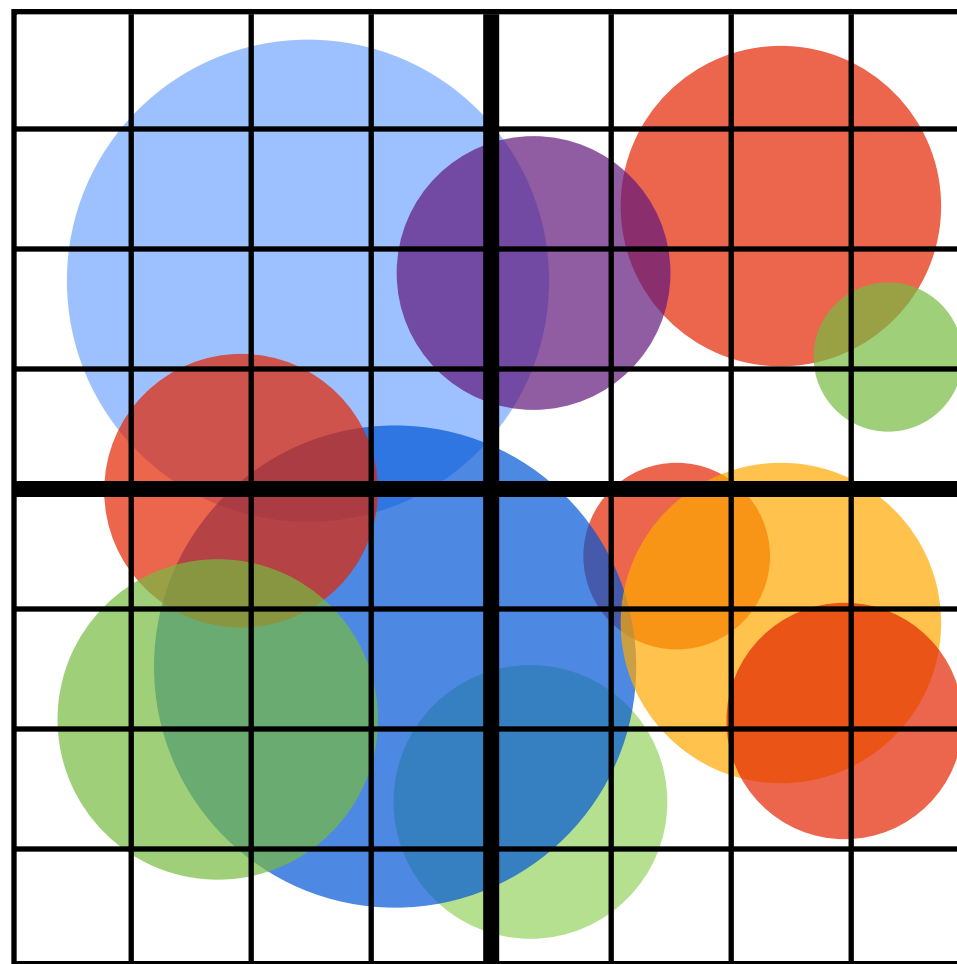


Example: scaling down assignment 2

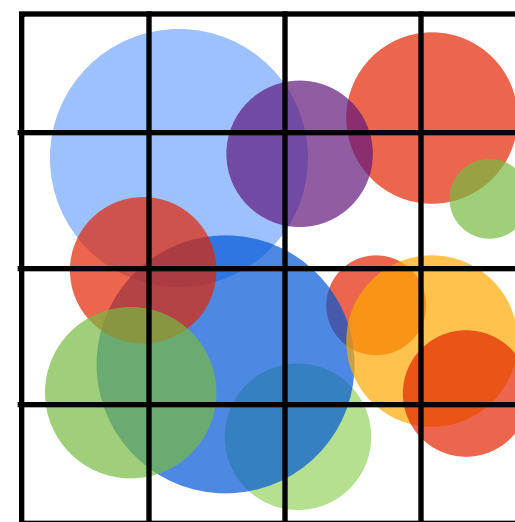
- Problem size = (w, h, num_circles, ...)

image width, height

number of scene circles

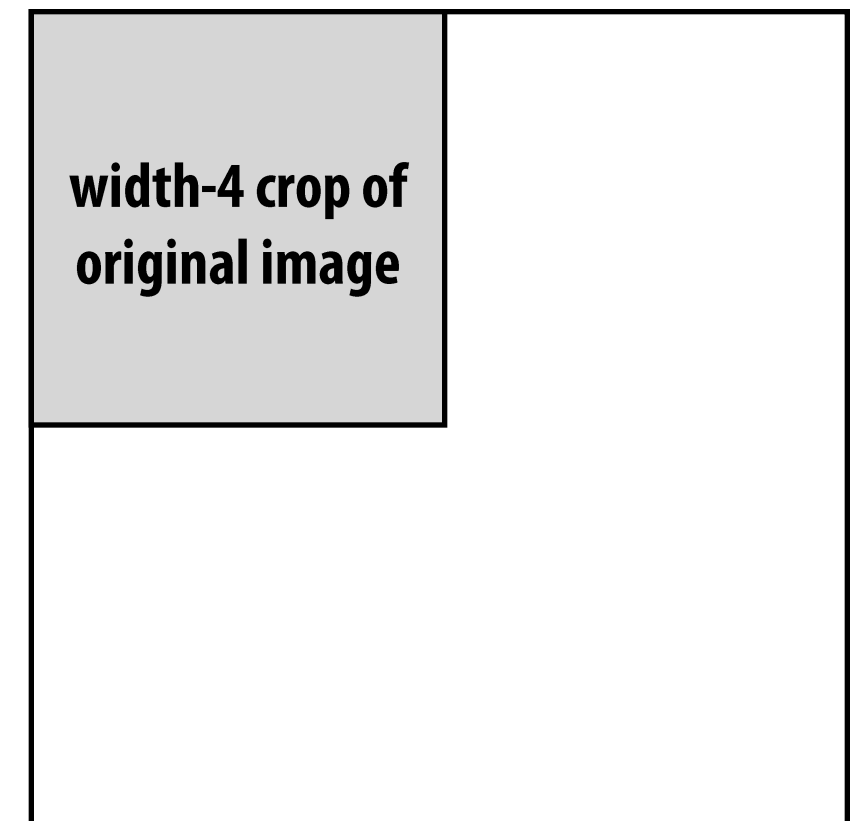


Original image
(w=8)



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

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.

**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?**

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

Scenario

- **Student walks into office hours and says, “My code is slow / uses lots of memory / is SIGKILLED. I implemented X, Y, and Z. Are those good? What should I do next?”**
- **It depends.**

What is my program doing?

- **Measurements are more valuable than insights**
 - **Insights are best formed from measurements!**
- **We're Computer Scientists**
 - **We can write programs to analyze programs**

Note about Examples

- **The example programs in today's lecture are from Spring 2016 Assignment 3**
 - **OpenMP-based graph processing workload (paraGraph)**
 - **Millions to tens of millions of nodes**
 - **Code written for the GHC machines and Xeon Phi**
- **Interesting example of parallel programming (bonus content)**

paraGraph

- **Optimize three graph kernels**
 - **bfs**
 - **kbfs – bfs k times, using different starting points**
 - **pagerank**

Mapping functions onto Graphs

- **Within OpenMP, but does so by mapping functions to vertices and edges**
 - **vertexMap(vertexMap(U: vertexSubset, F: vertex -> bool) : vertexSubset**
 - **edgeMap(G: graph, U: vertexSubset, C: vertex -> bool, F: (vertex, vertex) -> bool) : vertexSubset**

BFS

```
int* parents = ... initialized to -1
```

```
bool cond(int v) {  
    return parents[v] == -1;  
}
```

```
bool update(int u, int v) {  
    return compare_and_swap(& parents[v], -1, u);  
}
```

```
void bfs(graph g, int root, int* parents) {  
    parents[root] = root;    // parent of root vertex is itself  
    vertexSubset frontier = {root};  
    while (size(frontier) > 0) {  
        frontier = edgeMap(graph, frontier, cond, update);  
    }  
}
```

My program is slow today.

- What else is running?
 - Try “top”

```
top - 14:43:26 up 25 days, 3:46, 50 users, load average: 0.04, 0.05, 0.01
Tasks: 1326 total, 1 running, 1319 sleeping, 2 stopped, 4 zombie
Cpu(s): 0.0%us, 0.1%sy, 0.0%ni, 99.9%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 16220076k total, 7646188k used, 8573888k free, 246280k buffers
Swap: 4194296k total, 3560k used, 4190736k free, 5219176k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2801	nobody	20	0	481m	3860	1192	S	1.0	0.0	63:45.33	gmetad
3306	root	20	0	258m	11m	2128	S	0.7	0.1	161:54.86	lsi_mrdsnmpagen
4920	nobody	20	0	297m	18m	3380	S	0.7	0.1	181:11.80	gmond
49781	-----	20	0	106m	2144	1456	S	0.3	0.0	0:00.10	bash
58119	bpr	20	0	15976	2220	936	R	0.3	0.0	0:00.30	top
106182	-----	20	0	24584	2184	1136	S	0.3	0.0	2:27.99	tmux
134225	-----	20	0	143m	1732	608	S	0.3	0.0	0:02.92	intelremotemond
...											

What else can top tell us?

- **CPU / Memory usage of our program**

- `./paraGraph kbfs com-orkut_117m.graph -t 8 -r`

```
top - 15:54:27 up 3 days, 23:58,  6 users,  load average: 3.43, 1.15, 0.43
Tasks: 286 total,  2 running, 284 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.8 us,  0.2 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem: 32844548 total, 31305468 used,  1539080 free,   435012 buffers
KiB Swap:  7999484 total,   13176 used,  7986308 free. 27364456 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
23457	bpr	20	0	1559584	979704	3420	R	796.4	3.0	0:27.91	paraGraph
1071	root	20	0	75892	6560	5564	S	2.0	0.0	19:58.05	cups-brows+
21506	root	20	0	87680	17300	5460	S	0.7	0.1	1:08.43	cupsd
23408	bpr	20	0	24956	3196	2588	R	0.3	0.0	0:00.18	top
1	root	20	0	36100	4204	2632	S	0.0	0.0	0:01.02	init

Do I have to use top?

- **No. Time was part of the assignment 3 qsub jobs.**

```
$ tail -n 1 bpr_grade_performance.job  
time ./grade_performance.py ./$exe
```

- **time is often a shell command, there is also the time binary**

- ```
/usr/bin/time ./paraGraph kbfs com-orkut_117m.graph -t 8 -r
...
33.16user 0.10system 0:05.54elapsed 600%CPU (0avgtext+0avgdata
979708maxresident)k 0inputs+0outputs
(0major+5624minor)pagefaults 0swaps
```

# But why is it slow?

- **Where is the time spent?**
  - **Put timing statements around probable issues**
  - **Print results**
- **OR**
  - **Use a tool to insert timing statements**

# Program Instrumentation

- **When to inject the instrumentation?**
  - **When the program is compiled.**
  - **When the program is run.**

# Instrumentation Tool Families

- **Program Optimization**
  - **Gprof**
  - **Perf**
  - **VTune**
- **Program Debugging**
  - Valgrind
  - Sanitizers
- **Advanced Analysis**
  - Pin
  - Contech

# Amdahl's Law Revisited

- $1 - s$  – a component of the program
- $p$  – speedup of that component
- The more time something takes
  - The more speedup small improvements make
- Concentrate program optimization on:
  - Hot code
  - Common cases

$$\text{speedup} \leq \frac{1}{s + \frac{1-s}{p}}$$

# GProf

- Enabled with “-pg” compiler flag
- Places a call into every function
  - Calls record the call graph
  - Calls record time elapsed
- Run the program.
- Run `gprof <prog name>`

# GProf cont

- **Output shows both the total time in each function**
  - **And cumulative time in calling trees**
- **Can be useful with large call graphs**
- `$/paraGraph pagerank -t 8 -r soc-pokec_30m.graph`
- `$gprof`

| %<br>time | cumulative<br>seconds | self<br>seconds | calls   | self<br>ms/call | total<br>ms/call | name                         |
|-----------|-----------------------|-----------------|---------|-----------------|------------------|------------------------------|
| 69.35     | 0.43                  | 0.43            | 1       | 430.00          | 430.00           | build_incoming_edges(graph*) |
| 30.65     | 0.62                  | 0.19            | 18      | 10.56           | 10.56            | pagerank(graph*, ...)        |
| 0.00      | 0.62                  | 0.00            | 1632803 | 0.00            | 0.00             | addVertex(VertexSet*, int)   |
| 0.00      | 0.62                  | 0.00            | 7       | 0.00            | 0.00             | newVertexSet(T, int, int)    |
| 0.00      | 0.62                  | 0.00            | 7       | 0.00            | 0.00             | freeVertexSet(VertexSet*)    |

# Perf

- **Modern architectures expose performance counters**
  - **Cache misses, branch mispredicts, IPC, etc**
- **Perf tool provides easy access to these counters**
  - **perf list – list counters available on the system**
  - **perf stat – count the total events**
  - **perf record – profile using one event**
  - **perf report – Browse results of perf record**
- **Perf is present on GHC machines tested**

# Perf stat

- Can be run with specific events or a general suite
- **perf stat [-e ...] app**
  - Many counters come in pairs, each needs a separate -e
    - cycles, instructions
    - branches, branch-misses
    - cache-references, cache-misses
    - stalled-cycles-frontend
    - stalled-cycles-backend
  - Processors can only enable ~4 counters, else it must multiplex

# Perf stat (default) output

```
./paraGraph -t 8 -r pagerank /afs/cs/academic/class/15418-s16/public/asst3_graphs/soc-
pokec_30m.graph':
```

|               |                         |   |                              |          |
|---------------|-------------------------|---|------------------------------|----------|
| 2366.633970   | task-clock (msec)       | # | 1.758 CPUs utilized          |          |
| 109           | context-switches        | # | 0.046 K/sec                  |          |
| 9             | cpu-migrations          | # | 0.004 K/sec                  |          |
| 6,168         | page-faults             | # | 0.003 M/sec                  |          |
| 7,513,900,068 | cycles                  | # | 3.175 GHz                    | (83.23%) |
| 6,327,732,886 | stalled-cycles-frontend | # | 84.21% frontend cycles idle  | (83.42%) |
| 4,019,403,839 | stalled-cycles-backend  | # | 53.49% backend cycles idle   | (66.86%) |
| 3,222,030,372 | instructions            | # | 0.43 insns per cycle         |          |
|               |                         | # | 1.96 stalled cycles per insn | (83.43%) |
| 457,170,532   | branches                | # | 193.173 M/sec                | (83.30%) |
| 12,354,902    | branch-misses           | # | 2.70% of all branches        | (83.24%) |

So what is the bottleneck?

# More perf stat

- **Maybe memory is a bottleneck.**

|             |                  |   |        |                     |  |
|-------------|------------------|---|--------|---------------------|--|
| 201,493,787 | cache-references |   |        |                     |  |
| 49,347,882  | cache-misses     | # | 24.491 | % of all cache refs |  |

- **24% misses, that's not good.**
- **But what should we do?**

# Perf record

- **Pick an event (or use the default cycles)**
- **When the event's counter overflows**
  - **The processor sends an interrupt**
  - **The kernel records where (PC value) of the program**
- **NOTE: counters update in funny, microarchitectural ways so intuition may be required**

**“Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it.”**

# Perf cache misses

- Are cache misses the problem?
  - Sort of.

Samples: 11K of event 'cache-misses', Event count (approx.): 181771931

| Overhead | Command   | Shared Object | Symbol                      |
|----------|-----------|---------------|-----------------------------|
| 47.18%   | paraGraph | paraGraph     | [.] edgeMapS<State<float> > |
| 46.84%   | paraGraph | paraGraph     | [.] build_incoming_edges    |
| 2.70%    | paraGraph | [unknown]     | [k] 0xffffffff813b2537      |
| 1.37%    | paraGraph | [unknown]     | [k] 0xffffffff813b2915      |

# Perf report cycles

- **perf report shows analysis from record**
  - **Commandline interactive interface**

Samples: 13K of event 'cycles', Event count (approx.): 11108635969

| Overhead | Command   | Shared Object     | Symbol                       |
|----------|-----------|-------------------|------------------------------|
| 65.93%   | paraGraph | paraGraph         | [.] edgeMapS<State<float> >  |
| 27.66%   | paraGraph | paraGraph         | [.] build_incoming_edges     |
| 1.85%    | paraGraph | paraGraph         | [.] vertexMap<Local<float> > |
| 1.02%    | paraGraph | [kernel.kallsyms] | [k] clear_page_c             |
| 0.88%    | paraGraph | paraGraph         | [.] addVertex                |
| 0.60%    | paraGraph | [kernel.kallsyms] | [k] copy_user_generic_string |

- **Over 25% of program time is in creating the graph**
  - **This also skews the perf stats**

# Deep dive

- Selecting a function will display its assembly with function-local %

```

| bool update(Vertex s, Vertex d)
| {
| float add = pcurr[s] / outgoing_size(graph, s);
2.97 | divss %xmm1,%xmm0
5.22 | jmp 162
| nop
|160: mov %eax,%edx
| #pragma omp atomic
| pnext[d] += add;
0.16 |162: mov %edx,0x18(%rsp)
1.28 | mov %edx,%eax
0.01 | movss 0x18(%rsp),%xmm2
2.71 | addss %xmm0,%xmm2
4.63 | movss %xmm2,0x18(%rsp)
1.16 | mov 0x18(%rsp),%r15d
3.99 | lock cmpxchg %r15d, (%rcx)
25.22 | cmp %eax,%edx
| jne 160
```

# Deep dive

- Selecting a function will display its assembly with function-local %

```

| bool update(Vertex s, Vertex d)
| {
| float add = pcurr[s] / outgoing_size(graph, s);
2.97 | divss %xmm1,%xmm0
5.22 | jmp 162
| nop
|160: mov %eax,%edx
| #pragma omp atomic
| pnext[d] += add;
0.16 |162: mov %edx,0x18(%rsp)
1.28 | mov %edx,%eax
0.01 | movss 0x18(%rsp),%xmm2
2.71 | addss %xmm0,%xmm2
4.63 | movss %xmm2,0x18(%rsp)
1.16 | mov 0x18(%rsp),%r15d
3.99 | lock cmpxchg %r15d,(%rcx)
25.22 | cmp %eax,%edx
| jne 160
```

1. OMP atomic -> lock cmpxchg
2. This instruction is 25%\*65% of execution time

# Deep dive 2

- **kBFS is really, really slow. Why?**

Samples: 48K of event 'cycles', Event count (approx.):  
39218498652

| Overhead | Command   | Shared Object | Symbol                     |
|----------|-----------|---------------|----------------------------|
| 63.78%   | paraGraph | paraGraph     | [.] edgeMapS<RadiiUpdate>  |
| 19.33%   | paraGraph | paraGraph     | [.] edgeMap<RadiiUpdate>   |
| 8.21%    | paraGraph | paraGraph     | [.] build_incoming_edges   |
| 3.88%    | paraGraph | paraGraph     | [.] vertexMap<VisitedCopy> |

- **That's almost all my code. :(**
  - **edgeMap(S) is my code**

# Disassemble it!

## ■ What is taking all of kbfs's time?

```
bool update(Vertex src, Vertex dst) {
 | bool changed = false;
 | for (int j = 0; j < NUMWORDS; j++) {
 | if (visited[dst][j] != visited[src][j]) {
0.11 | mov 0x0(%r13),%rax
0.21 | mov (%rax,%rdi,1),%rbp
0.20 | mov (%rax,%rcx,8),%rax
14.88 | mov 0x0(%rbp),%ebp
1.15 | mov (%rax),%eax
68.27 | cmp %eax,%ebp
0.02 | je 108
 | // word-wide or
 | __sync_fetch_and_or(&(nextVisited[dst][j]), visited[dst]
1.54 | mov 0x8(%r13),%rcx
0.34 | or %eax,%ebp
0.02 | mov (%rcx,%rdi,1),%rcx
0.31 | lock or %ebp, (%rcx)
 | int oldRadius = radii[dst];
 | if (radii[dst] != iter) {
6.45 | mov 0x18(%r13),%ebp
```

# 2D Arrays

- `visited[dst][j]`
  - `visited` is `int**`
  - `dst` in  $O(\text{Nodes})$
  - `j` is  $O(1)$  (often  $\leq 4$ )
- What if `visited` was `int*[4]`?
  - Eliminate one memory operation

# Summary Questions

- **If you may have a performance issue:**
    - **Is the issue reproducible?**
      - **Do you have a workload?**
      - **Is the system stable?**
    - **Is the workload at full CPU?**
      - **If not, are there other users / processes running?**
      - **Or does the workload rely heavily on IO?**
    - **Is the CPU time confined to a small number of functions?**
      - **What is the most time consuming function(s)?**
      - **What is their algorithmic cost and complexity?**
- time / top
- gprof / perf

# Summary Continued

- **You have a reproducible, stable workload**
  - **The machine is otherwise idle**
  - **The workload is fully using its CPUs**
  - **The algorithms are appropriate**
- **Is there a small quantity of hot functions?**
  - **Are their cycles confined to specific functions?**
  - **Are the costs of the instructions understood?**

↑  
perf / VTune  
↓

# Instrumentation Tool Links

- Gprof - <https://sourceware.org/binutils/docs/gprof/>
- Perf - [https://perf.wiki.kernel.org/index.php/Main Page](https://perf.wiki.kernel.org/index.php/Main_Page)
- VTune - <https://software.intel.com/en-us/qualify-for-free-software/student>
- Valgrind - <http://valgrind.org/>
- Sanitizers - <https://github.com/google/sanitizers>
- Pin - <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>
- Contech - <http://bprail.github.io/contech/>

# Other links

- **Performance Anti-patterns:**

**<http://queue.acm.org/detail.cfm?id=1117403>**