CSEP 524: Parallel Computation

Brad Chamberlain

Tuesdays 6:30 – 9:20

MGH 231

What is parallel computing?

Parallel Computing:

What is parallel computing?

Parallel Computing: Using multiple compute resources to execute a computation

- typically processors and their memories

Why would we do this?

What is parallel computing?

Parallel Computing: Using multiple compute resources to execute a computation

- typically processors and their memories

Why would we do this?

- To execute a computation more quickly than we could otherwise
- Or to execute a larger program (in terms of amount of memory required, e.g.)

Parallel Computing for My Grandmother

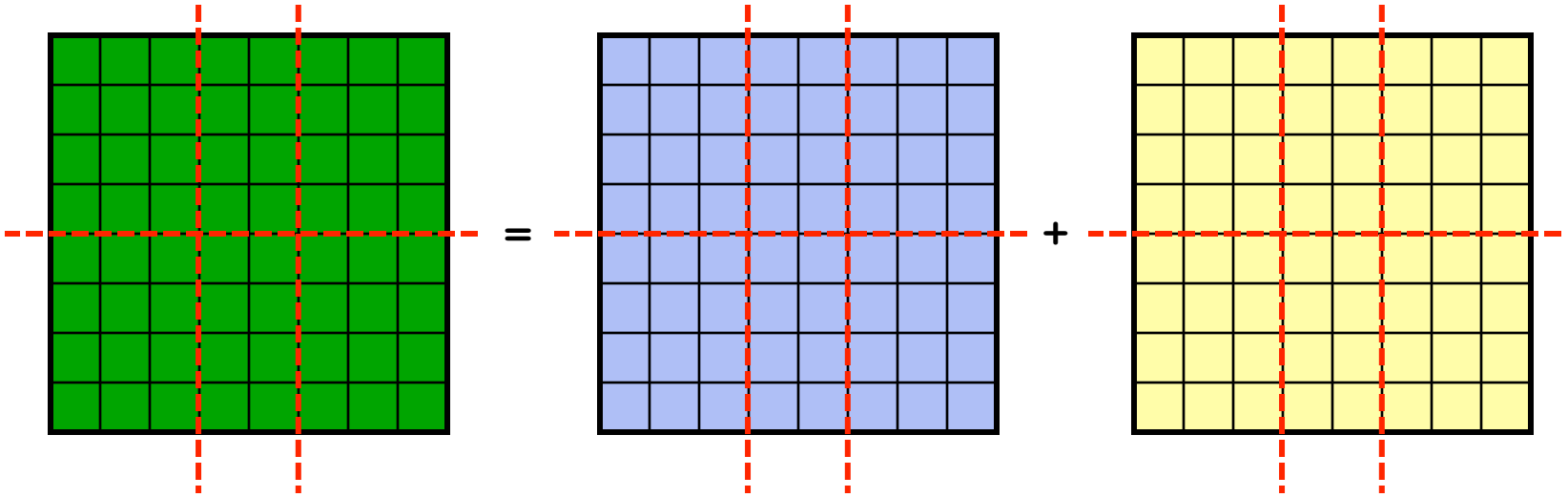
"Let's say you had some task which was going to take you 100 hours to finish. Imagine enlisting 100 friends to help you with the task. You now have some hope of finishing the task in 1 hour."

Is this realistic?

- It depends on the task…

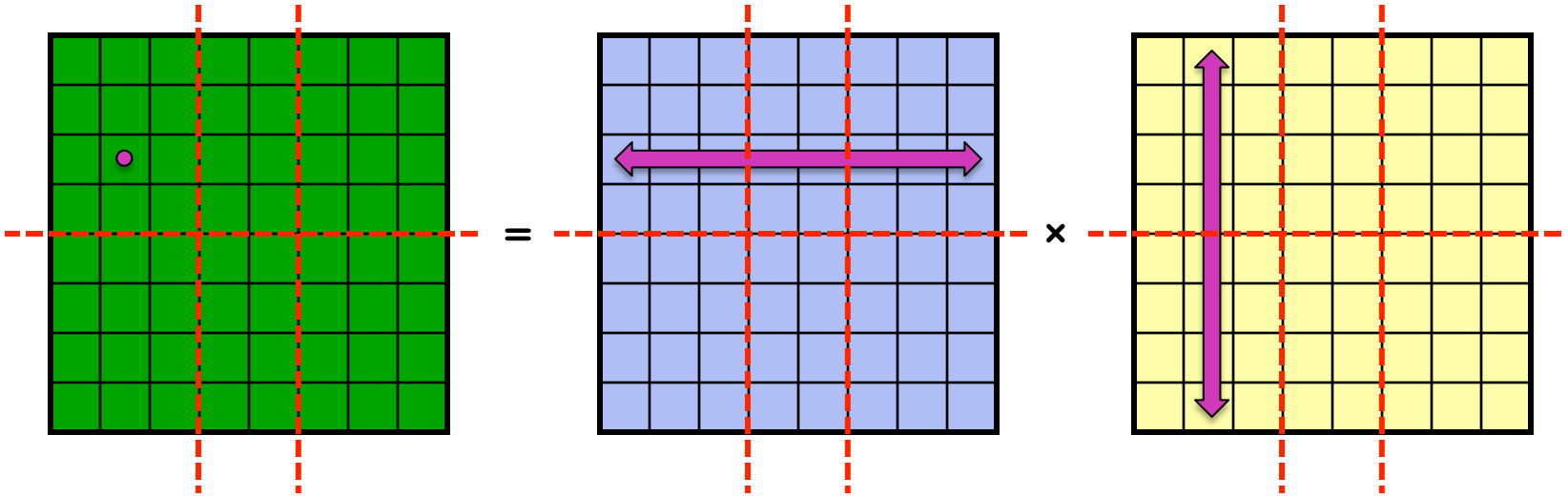
Parallel Computations Vary in Difficulty

Matrix Addition: Quite straightforward



Parallel Computations Vary in Difficulty

Matrix Multiplication: Far more involved



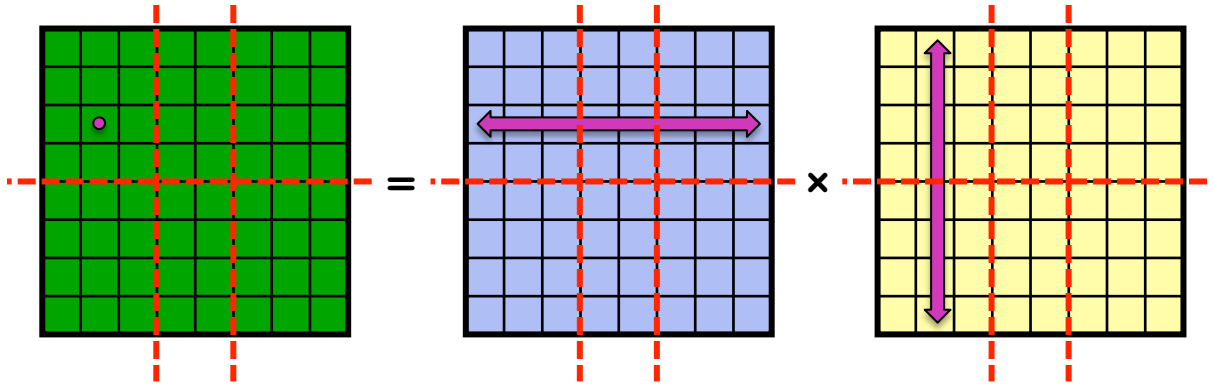
Parallel Computing: Two Key Concerns

Parallelism: "What should execute simultaneously?"

- without parallelism, no speed improvement from parallel computing

Locality: "Where should things execute?"

- attention to locality often necessary for top performance



Parallel Computing vs. Concurrency

Parallel Computing:

Concurrency:

Parallel Computing vs. Concurrency

(In a room of experts, definitions of these terms will vary greatly; these are the ones I prefer)

Parallel Computing:

- Typically done for performance reasons
- When parallelism ignored, program is still be correct
  - (just slower)

Concurrency:

- parallelism is intrinsically required (e.g., for correctness)
- simultaneous execution may be actual or virtualized

Parallel Computing: Related Terms

Parallel Programming: Writing programs that will execute in parallel

- (this will be a primary focus for this course)

Parallel Programming Models: A blanket term I use to refer to the languages, libraries, and pragmas used to express parallel programs

- "parallel programming notations" would be more precise

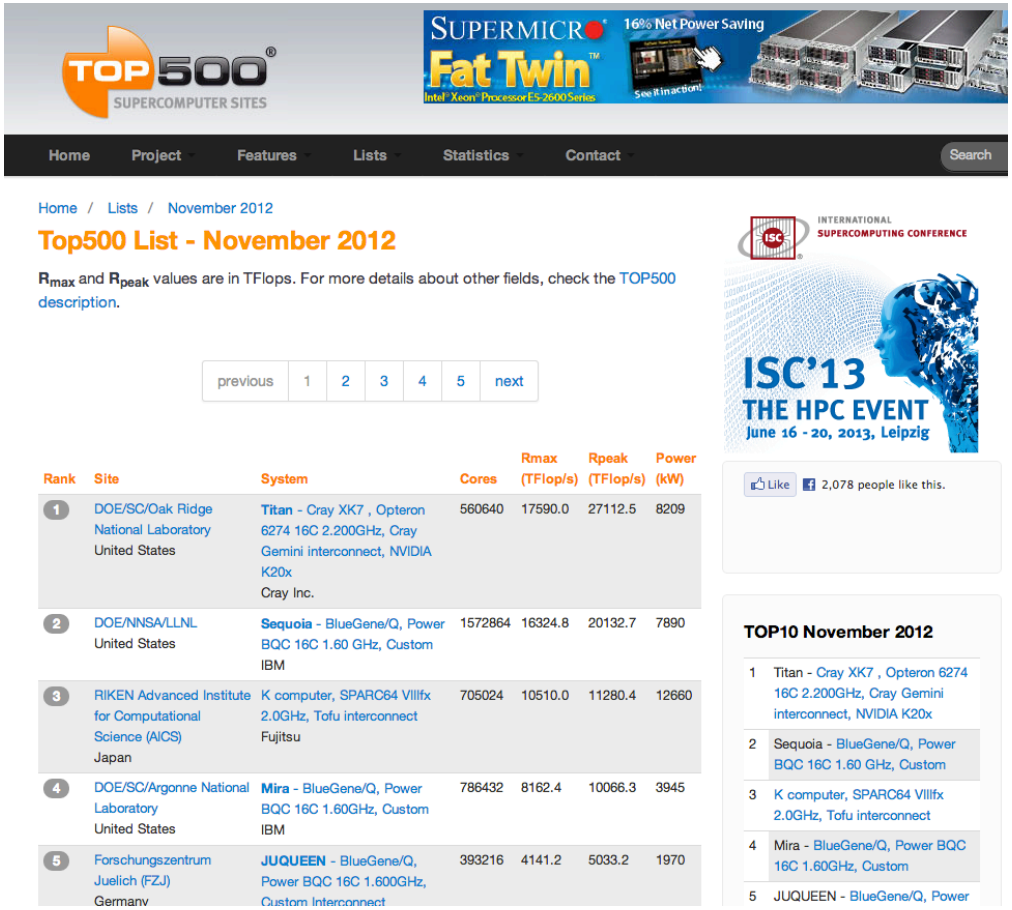
High-Performance Computing (HPC): Parallel computing on very large-scale systems

- also referred to as *Supercomputing* or *High-End Computing*

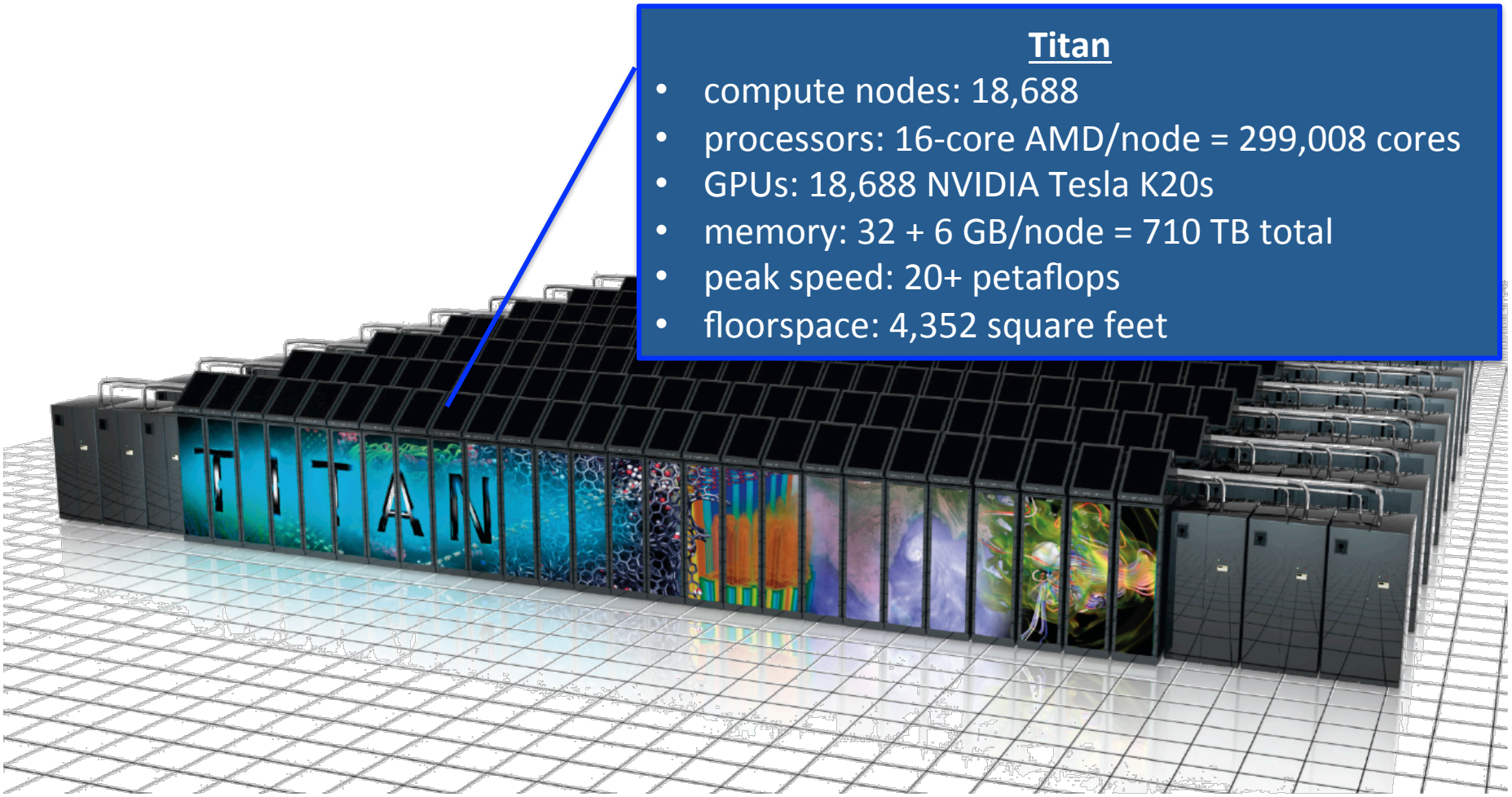
My Employer: CRAY THE SUPERCOMPUTER COMPANY

Top500: One way to compare supercomputers

- Rates the 500 fastest computers twice a year
- Measured using the LINPACK benchmark
  - Solves an LU factorization
  - Flops dominate runtime
- Yet, other factors limit most real applications
  - e.g., memory bandwidth



TITAN (Currently #1 on the Top 500)



Titan

- compute nodes: 18,688
- processors: 16-core AMD/node = 299,008 cores
- GPUs: 18,688 NVIDIA Tesla K20s
- memory: 32 + 6 GB/node = 710 TB total
- peak speed: 20+ petaflops
- floorspace: 4,352 square feet

For more information: http://www.olcf.ornl.gov/titan/

"Glad I'm not an HPC Programmer!"

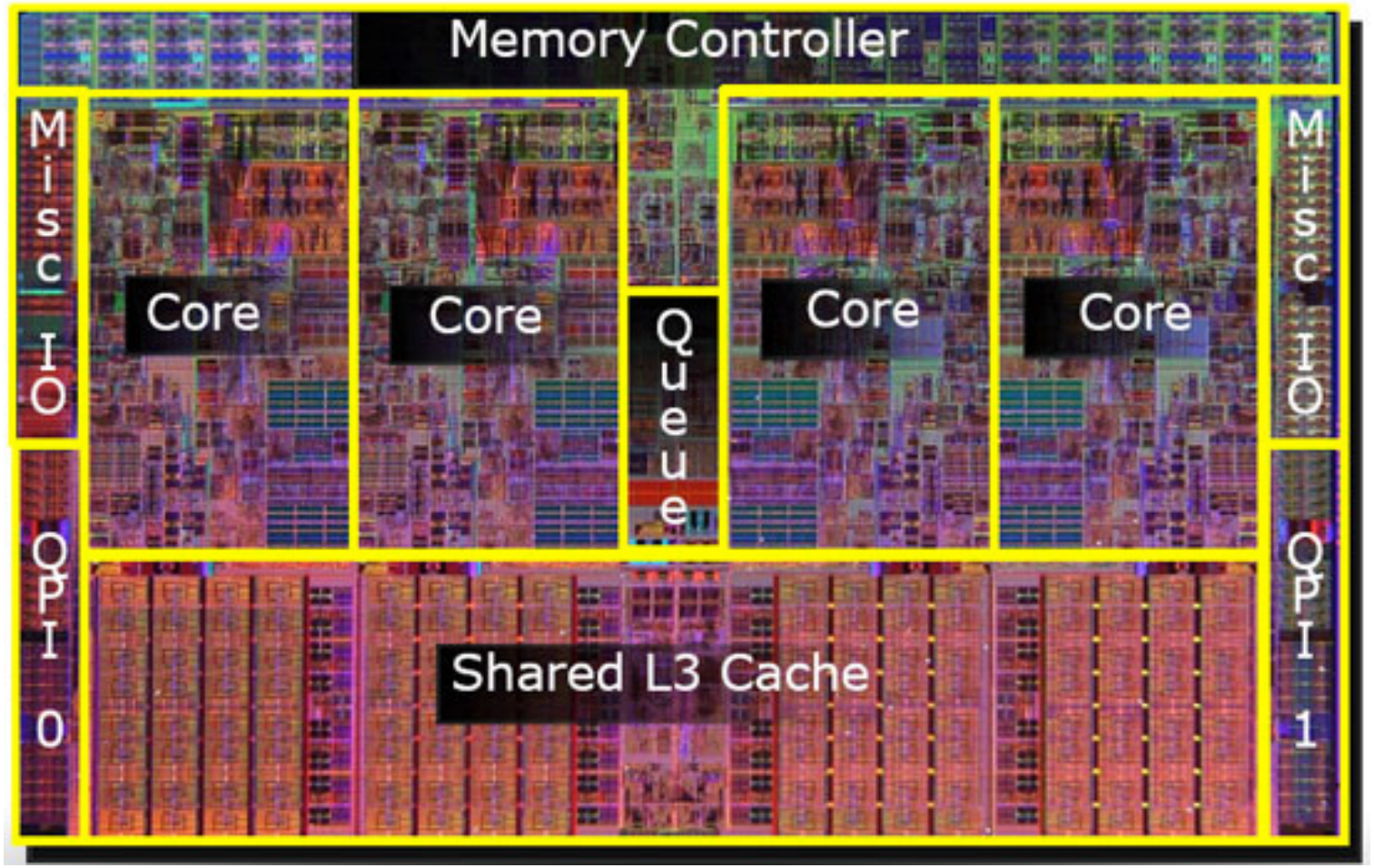
A Possible Reaction:

"This is all well and good for HPC users, but I'm a mainstream desktop programmer, so this is all academic for me."

The Unfortunate Reality:

- Performance-minded mainstream programmers will increasingly deal with parallelism
- And, as chips become more complex, locality too

My Mac's Processor: an Intel Core i7



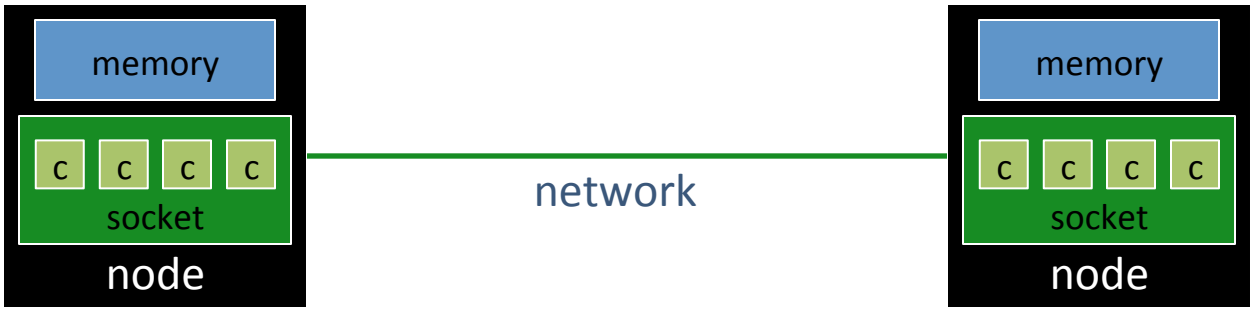
Source: http://www.legitreviews.com/article/1484/1/

Some Hardware Terminology

processor core (or simply "core"): the unit of a computer that has a PC, executes instructions, etc.

(compute) node: a group of cores and memories that must go over a network to communicate with any others

network: the wires and chips that permit nodes to communicate with one another



More HW Terms: Shared vs. Distributed Memory

***shared memory*:**

***distributed memory*:**

More HW Terms: Shared vs. Distributed Memory

shared memory: A system in which memory can be accessed via simple load/store instructions

- **example:** your multicore laptop/desktop
- also used to refer to programming models for such architectures
- typically executes a single OS image

distributed memory:

More HW Terms: Shared vs. Distributed Memory

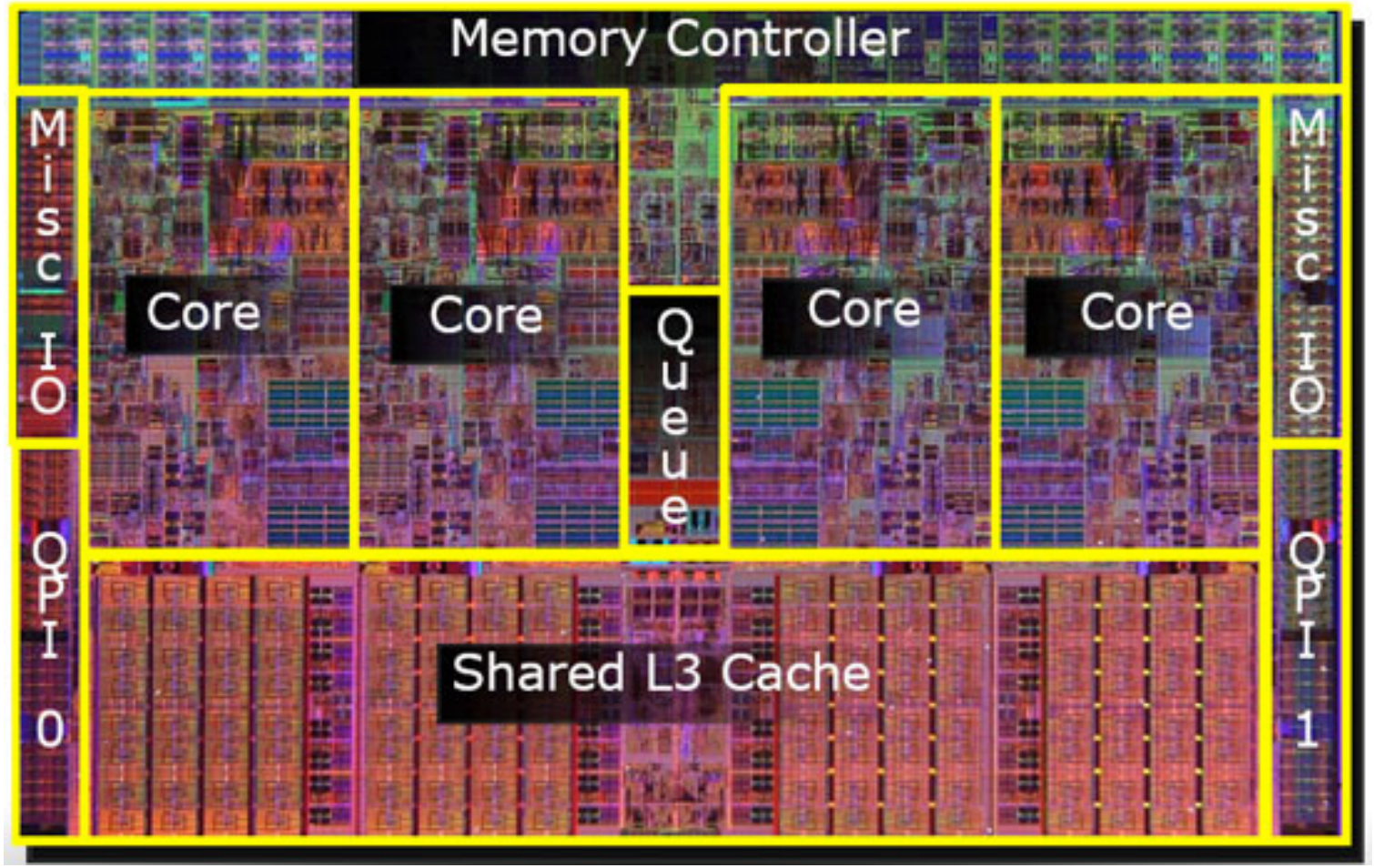
shared memory: A system in which memory can be accessed via simple load/store instructions

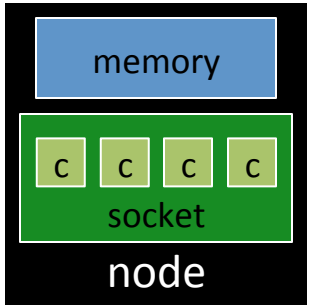
- **example:** your multicore laptop/desktop
- also used to refer to programming models for such architectures
- typically executes a single OS image

distributed memory: A system with multiple distinct memory segments that are not trivially accessible from one another

- **examples:** commodity clusters; desktops on a network
- typically an OS image per segment

For now, we'll focus on shared memory

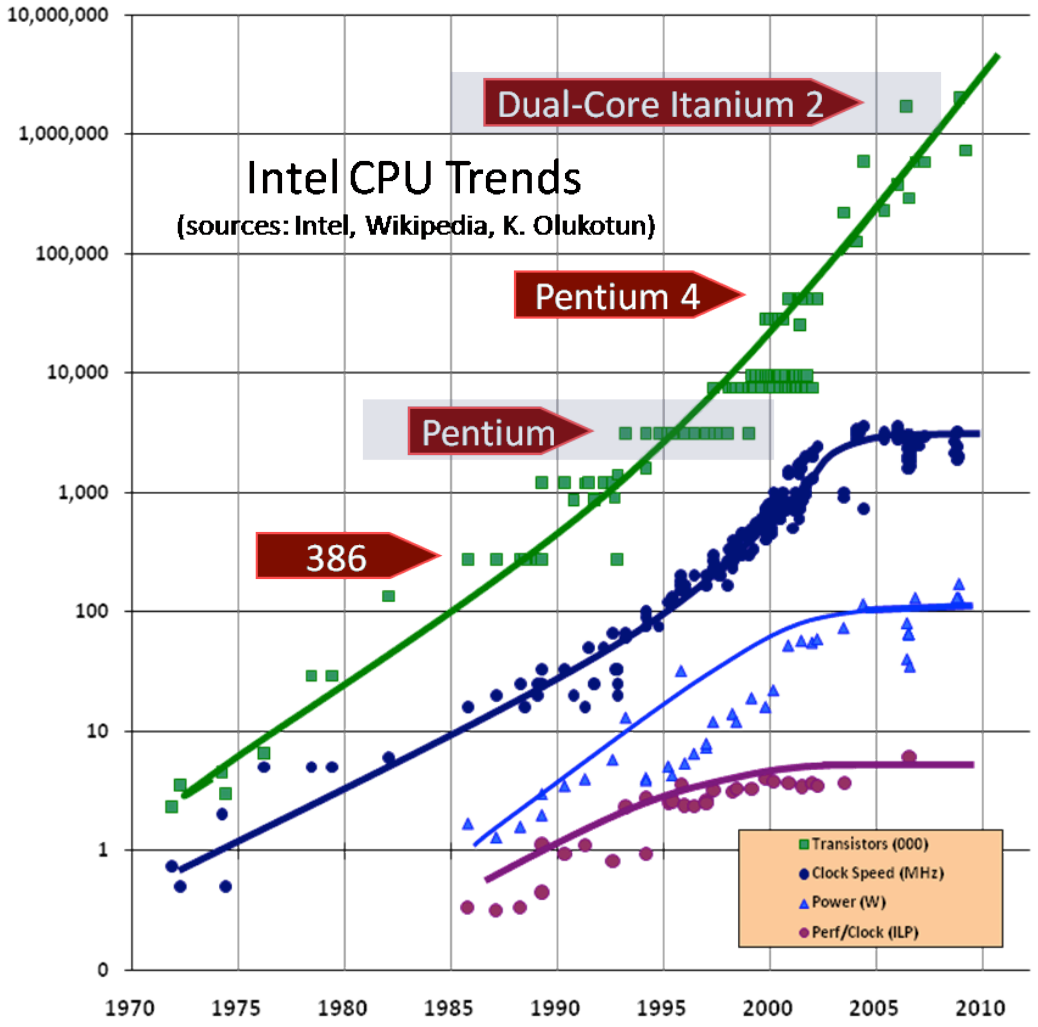




Source: http://www.legitreviews.com/article/1484/1/

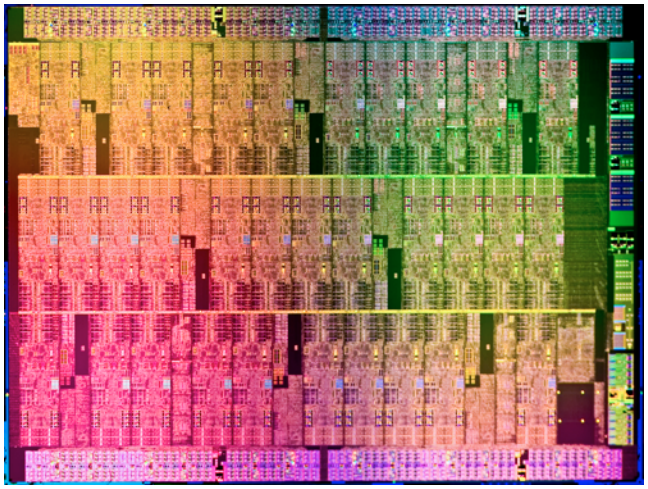
Multicore Processors: How did we get here?

- In short, transistor density has continued following Moore's Law
- But clock speeds have stopped increasing as rapidly as historically
- So what to do with extra transistors? And how to provide the performance boosts we're used to?
- Answer: add parallelism

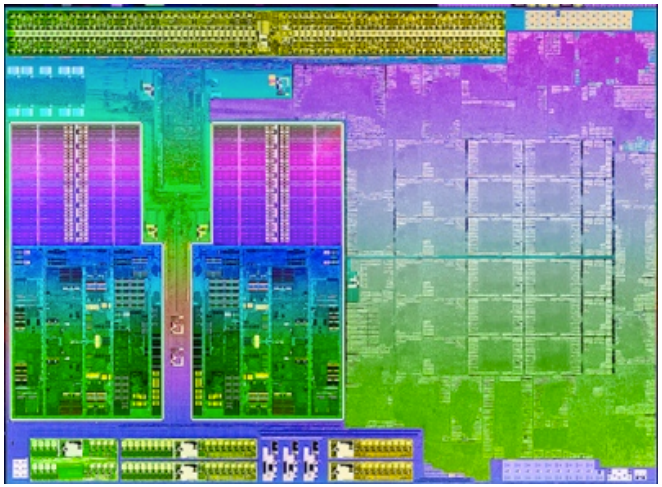


Source: *The Free Lunch is Over*, Herb Sutter, http://www.gotw.ca/publications/concurrency-ddj.htm

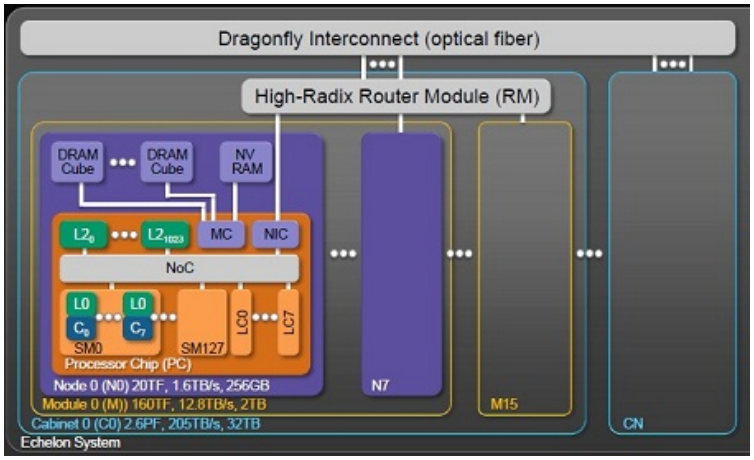
Prototypical Next-Gen Processor Technologies



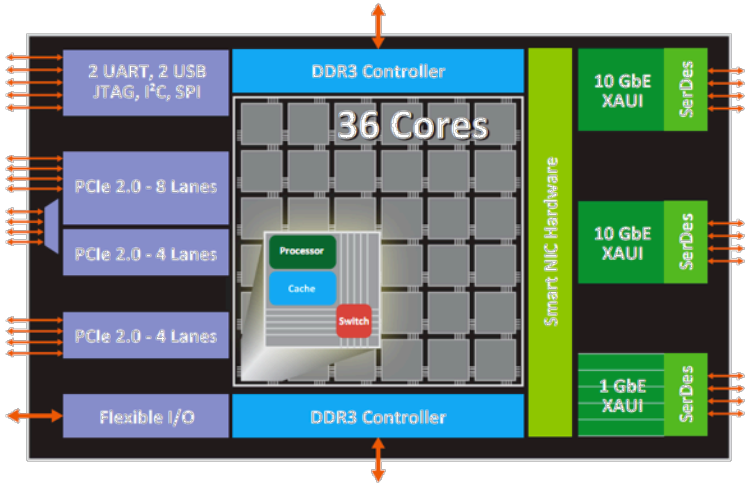
Intel MIC



AMD Trinity



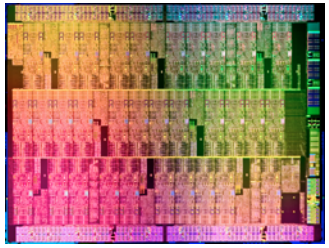
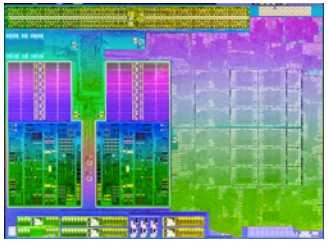
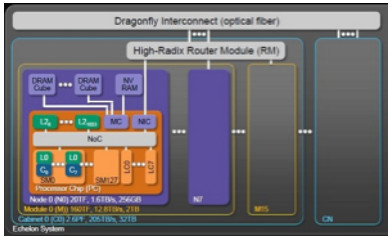
Nvidia Echelon

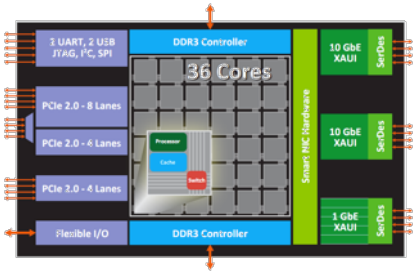


Tilera Tile-Gx

Sources: http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/
http://insidehpc.com/2010/11/26/nvidia-reveals-details-of-echelon-gpu-designs-for-exascale/ http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf

General Characteristics of These Architectures



- Increased hierarchy and/or sensitivity to locality
- Potentially heterogeneous processor/memory types
- Increasingly resemble supercomputers-on-a-chip

⇒ Next-gen programmers will have a lot more to think about at the node level than in the past

Outline for Today

- ✓ Parallel Computing: Definition and Motivation
- Introductory stuff
  - A little about me
  - Course overview
- Metrics for parallel execution
- **Algorithm:** Embarrassingly Parallel Computation
- **Programming Models:**
  - POSIX threads
  - Chapel
- This week's assignment

Me

My Parallel Computing Background

Graduated from UW in 2001

- dissertation focused on ZPL
  - an array-based parallel language developed at UW during the 1990's
- advisor: Larry Snyder
  - he taught this course until he retired
  - he also co-authored our textbook

Spent an educational year at a start-up

- worked on a parallel language for embedded computing

Joined Cray Inc. in 2002

- have worked primarily on Chapel
  - an emerging parallel language
  - will be used throughout this class

Supercomputing vs. Desktop Parallel Computing

- Most of my experience is in the HPC space
  - you may not particularly care about HPC

- Don't let this dissuade you from taking this class
  - at any scale, parallelism and locality are the key issues
  - the mainstream increasingly resembles HPC as time passes
  - thus, lessons learned from HPC will still apply

- Also, note that the mainstream is often HPC-like:
  - cloud computing, big data centers, map-reduce, etc.

Why am I teaching this class?

- As a favor to the department
- To put my money where my mouth is

  (we've been claiming that Chapel is an ideal language for parallel programming education for the past several years)
- To expose local professionals and students to Chapel

  (and get your feedback)
- As a personal challenge

The Good, the Bad (and hopefully no Ugly)

The Good:

- Having worked in this field for 20 years, I know it well
  - that said, there's also plenty I don't know
  - but I typically know who to ask/where to go to find answers
- I'm passionate about the subject matter

The Bad:

- This is my first time teaching a class in 13 years
  - will need your patience and feedback in throttling the pace
- And, it's my first time teaching this class
  - my slides are likely to be of varying quality and from various sources
- Like you, I have a day job

The Course

Overall Goals

- Expose you to as much information about parallel computing as possible within the allotted time
  - foundations
  - best practices
  - recent trends
- Teach you principles of parallel programming
- Give you practical parallel programming experience
  - using adopted programming models
    - Pthreads, OpenMP, MPI, UPC
  - using Chapel as an idealized parallel language

Course Content

Backbone: follow a progression of architectures and programming models from shared memory to distributed memory

Along the way: cover common parallel algorithms/ patterns, hazards, grab-bag topics, …

Class Sessions

- During the quarter, I hope to use a mix of:
  - mini-lectures
  - discussion
  - interactive programming
  - Q&A
  - guest speakers
  - inspirational music?, jokes?, …?

…to break up the 3-hour timeslot.

- You can help prevent things from dragging by asking questions, participating in discussion, etc.

Your Work

Assignments:

- **format:** A mix of readings, programming problems, and thinking/writing questions
- **rate:** Due every 1-2 weeks
- **late policy:** Twice during the quarter, you may turn in an assignment late (intended for use with work/family emergencies)

End-of-term project:

- learn about and report on some technology we didn't cover
- or possibly an independent programming project
  - challenge: picking a scope that's neither too hard nor too easy

Course Software

- We'll be using the departmental Fedora 17 VM as our official OS
  - the one we're committed to answering questions about
  - the one we'll use for executing your code in grading

- You can choose to use your own OS at your own risk
  - Linux/Mac/BSD – will *probably* work just fine
  - Windows – ???

Nuts and Bolts

TA: Brandon Myers (may also add a second ½-time TA)

Text: Lin & Snyder, *Principles of Parallel Programming*

(2nd printing contains a number of bug fixes)

Office Hours:

- Brandon and me: "office happy hour" just after class
- Brandon: Thursdays 1:30-2:30pm (CSE 218 and online)

Online Resources:

webpage (http://www.cs.washington.edu/education/courses/csep524/13wi/)

hangout, dropbox, feedback mechanisms (see www above)

Slides: will be made available in PDF (typically after lecture)

Questions for you

- Is anyone uncomfortable with C / UNIX?
- Anything else that you'd specifically like to see covered?
- Exam? Meet exam week?
- Other thoughts/questions/feedback?

(there will also be a survey covering these things)

Metrics for Parallel Execution

Scalability

Scalability:

Scalability

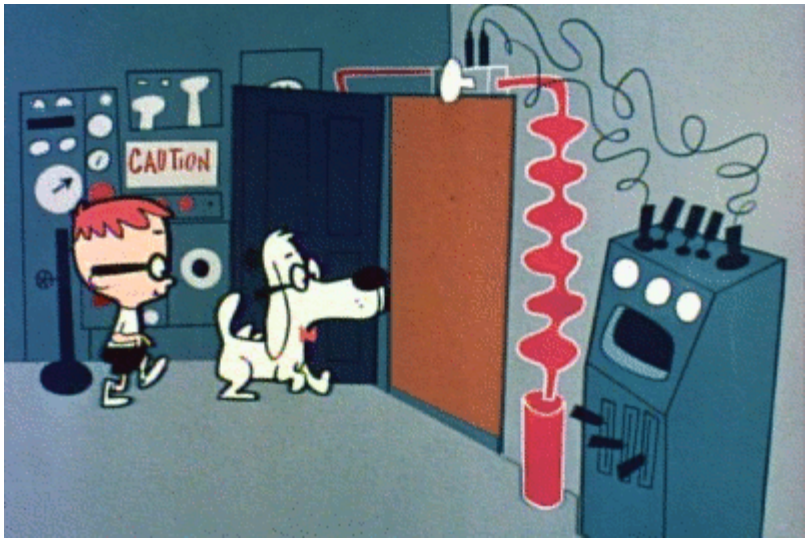
Scalability: the degree to which a program behaves well as we increase the scale at which it's executing

- "scale" typically refers to the number of processors
- it could also refer to other considerations like problem size

Ideally, parallel programs should be written to scale arbitrarily

- i.e., "That worked well on 8 processors, but now that we're running on 16 it falls over!" isn't very satisfying
  - in part to make it portable across architectures
  - in part because that's your route to better performance over time in a world without clock speed improvements

“Sherman, set the WABAC machine for 1999, CSE 373 (Data Structures and Algorithms)...”

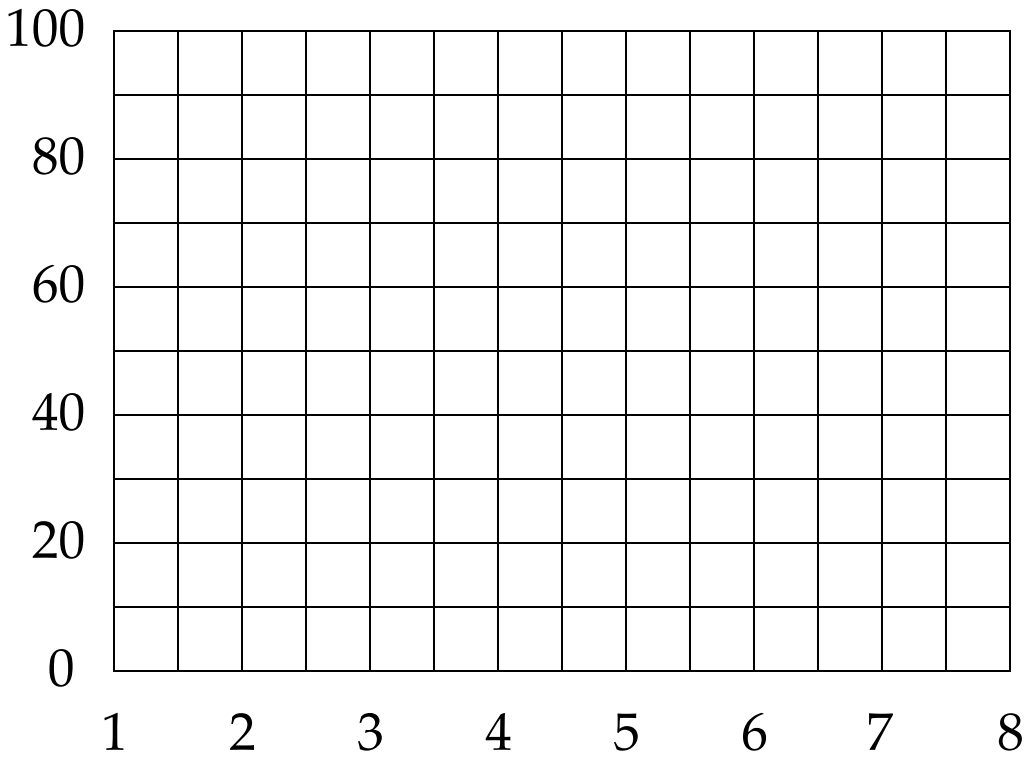


Simplifying Assumption

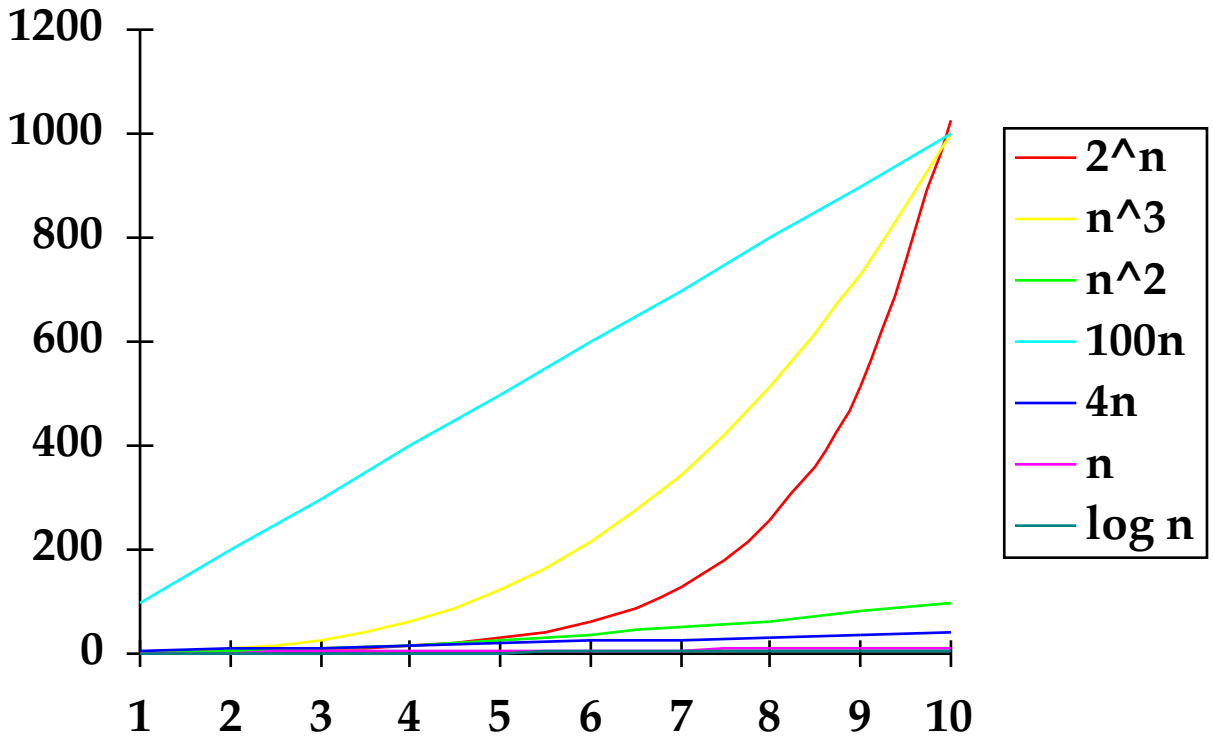
Constants are insignificant compared to the *asymptotic* behavior of the program

- expressed as a function of the problem size
- expressed using functions like: n, n^2, $\log n$, 2^n, etc.

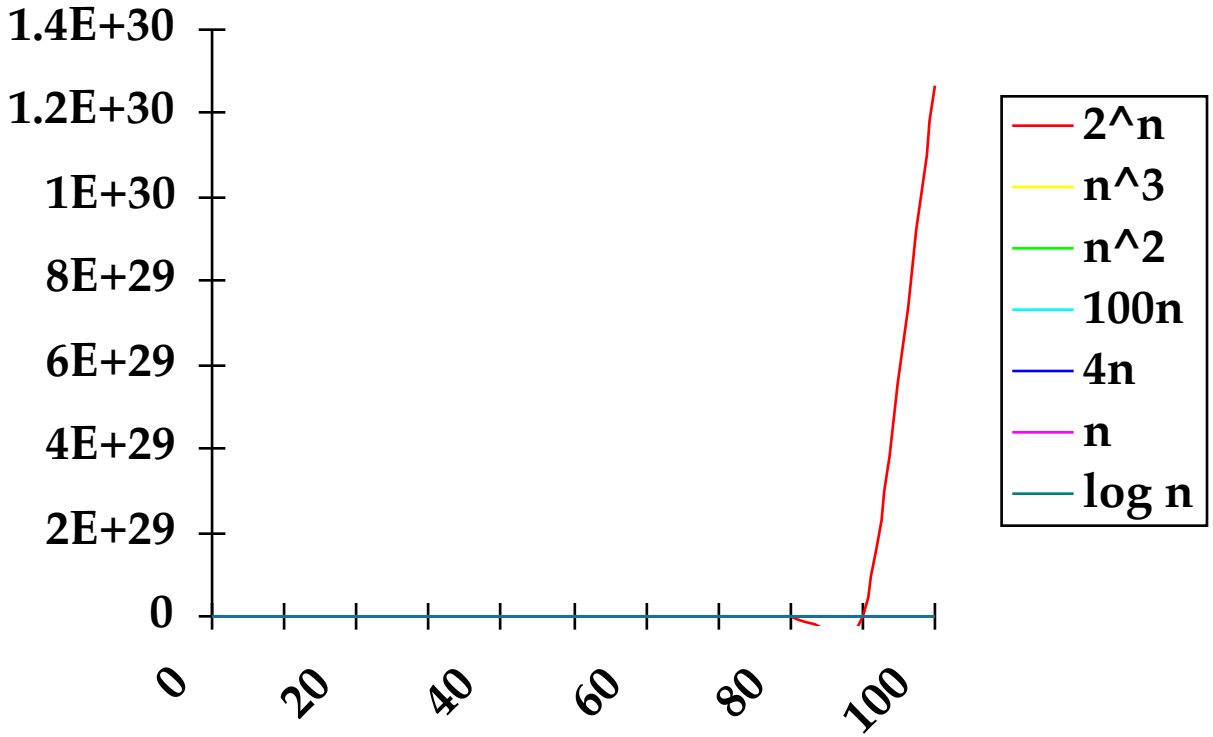
Getting some Intuition...



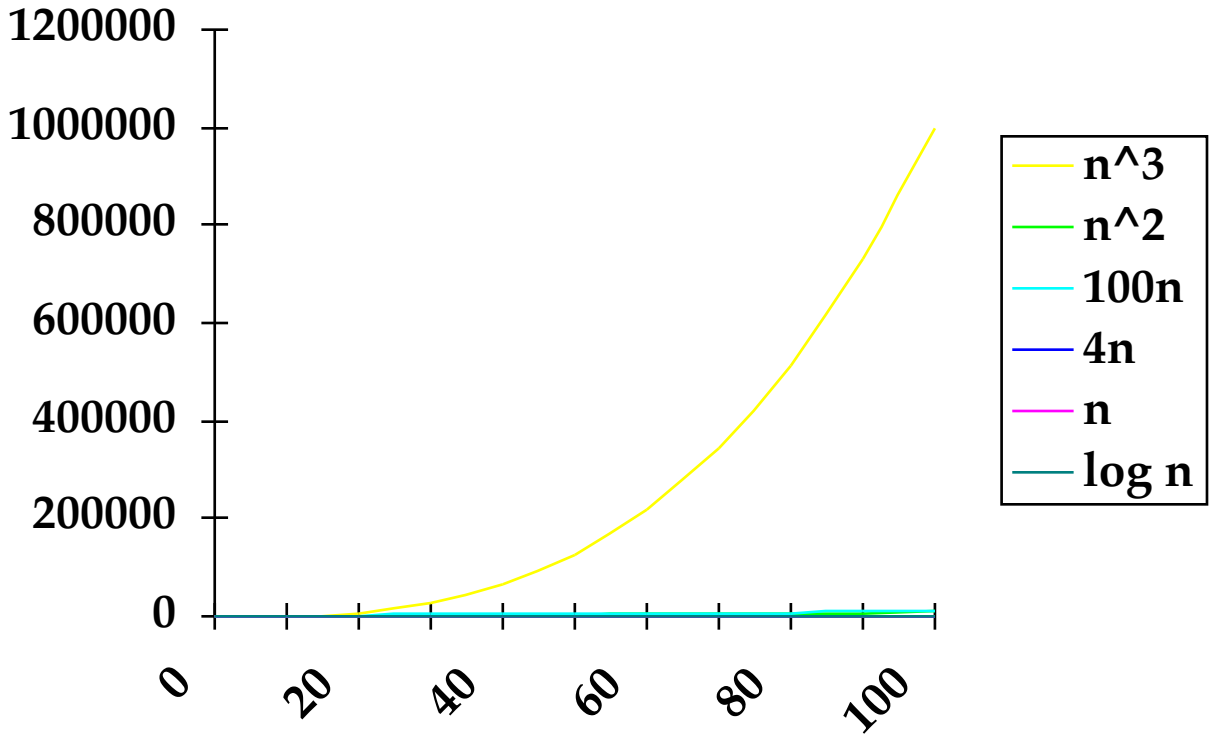
Using the Computer...



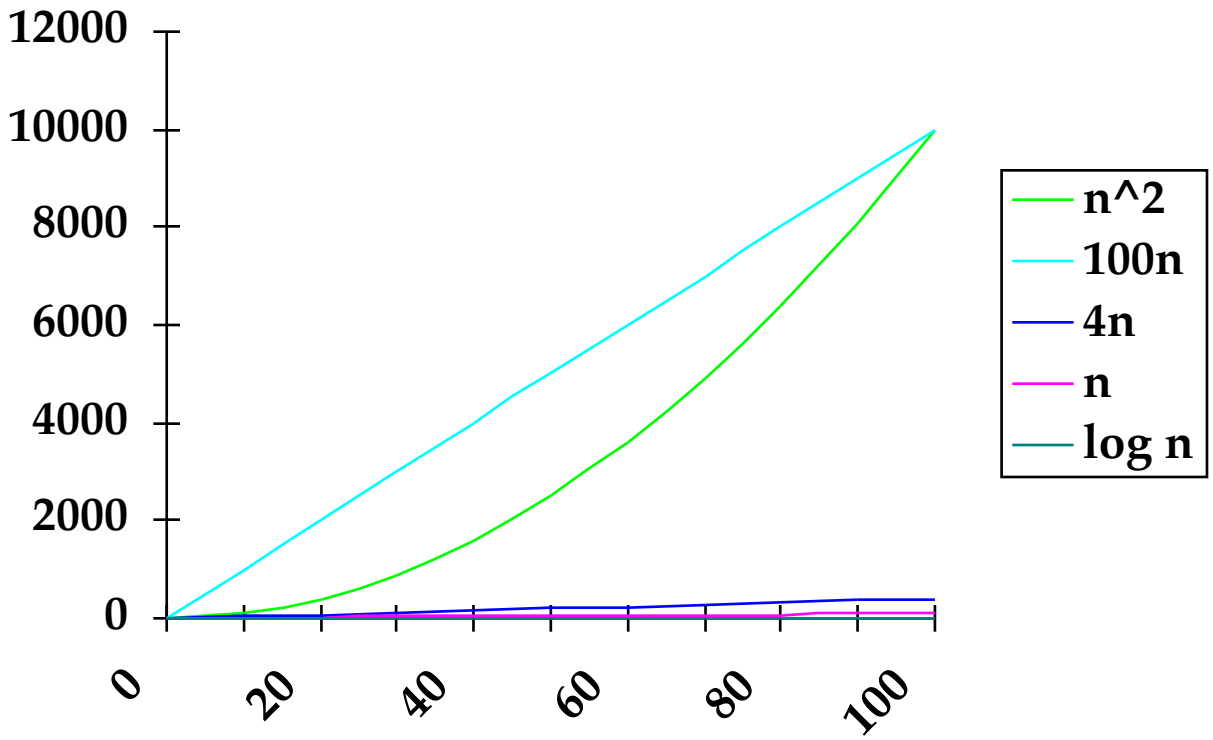
On A Larger Scale...



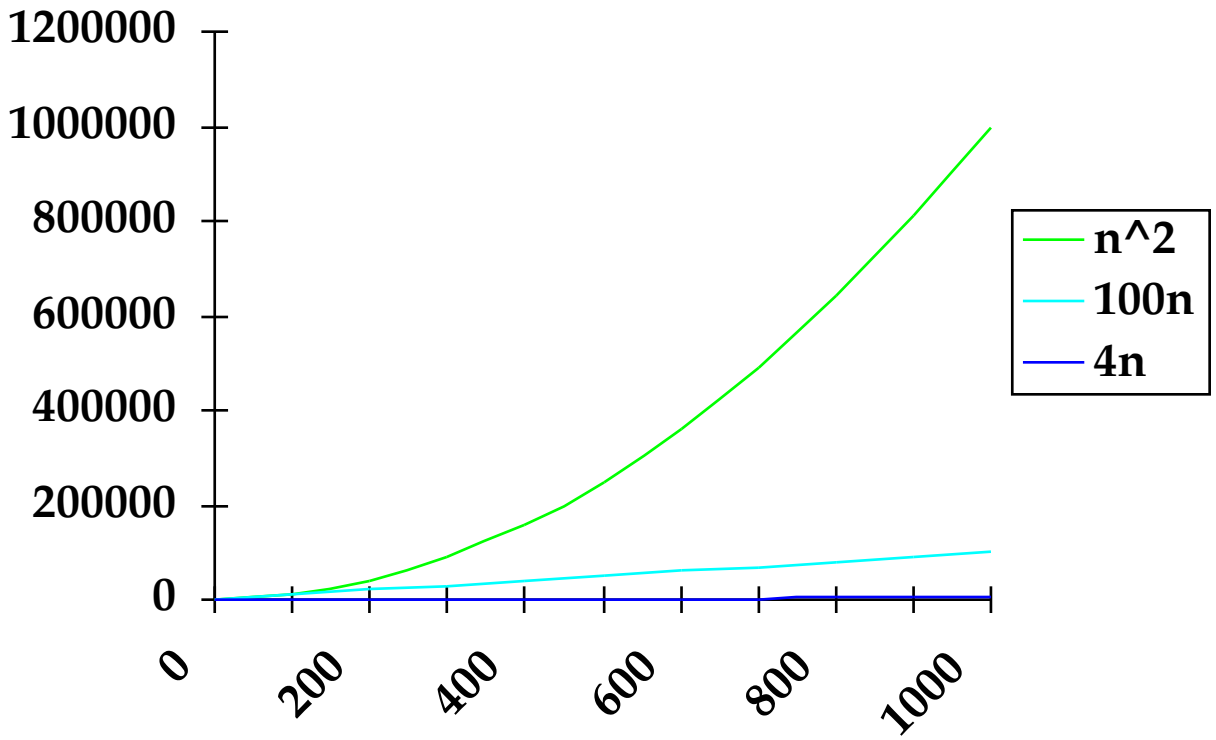
Ignoring 2^n


1200000
1000000
800000
600000
400000
200000
0
0
20
40
60
80
100
n^3
n^2
100n
4n
n
log n

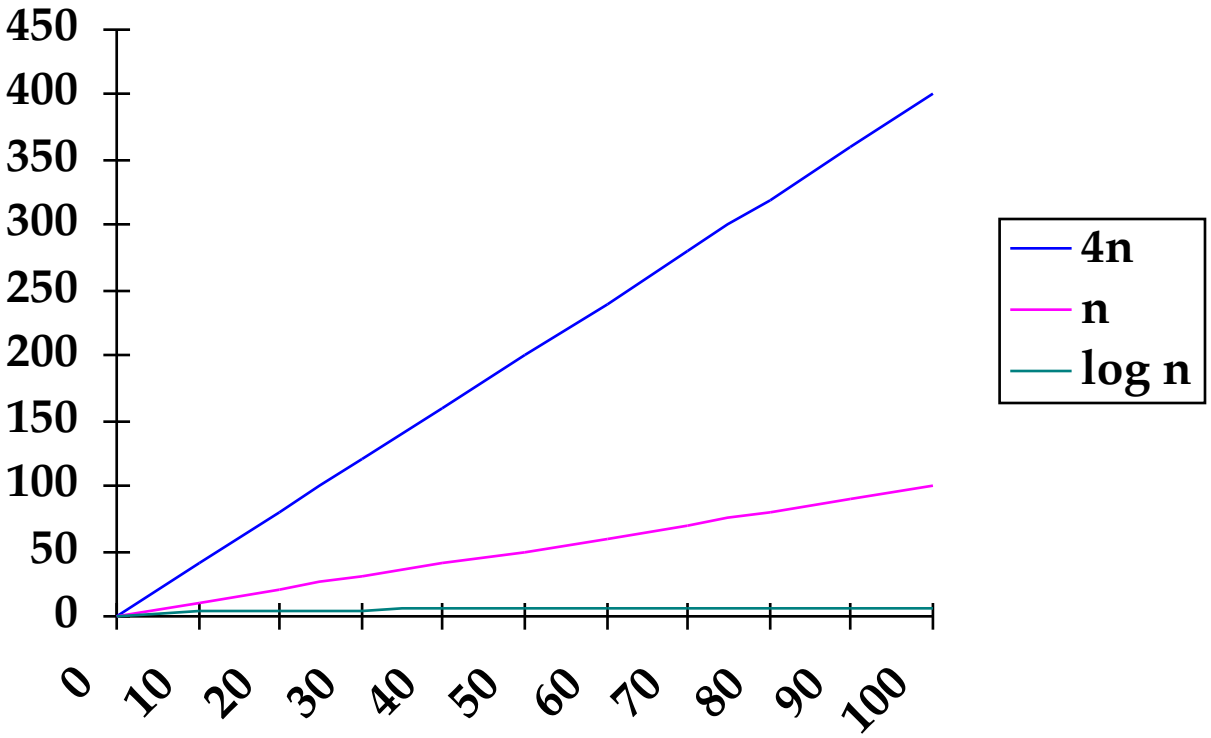
Ignoring n^3


12000
10000
8000
6000
4000
2000
0
0
20
40
60
80
100
n^2
100n
4n
n
log n

On Yet a Larger Scale



Smallest Functions Only



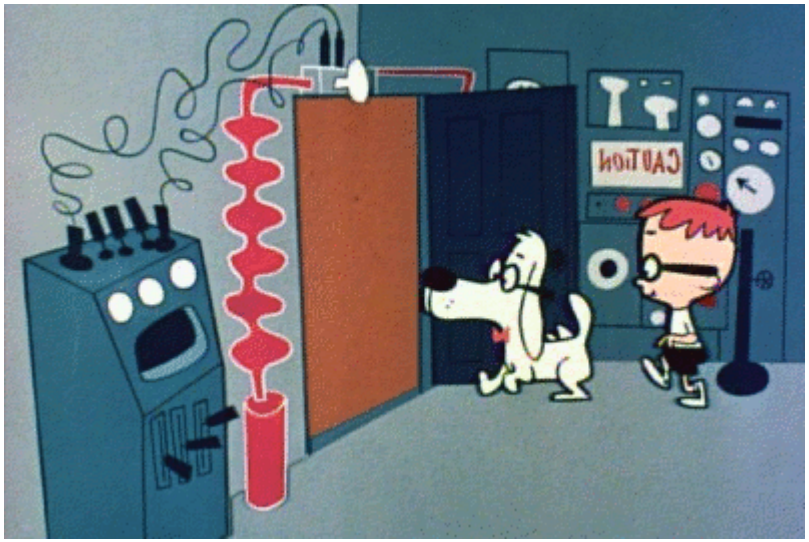
The Moral

Performance can be broken down into *primary* and *secondary effects*

- *primary effects*: asymptotic growth pattern
- *secondary effects*: constant factors, less significant terms

- In this class, we'll mainly be concerned with primary effects (*asymptotic analysis*)
- In the real world, secondary effects are also often worth paying attention to (*after* the primary ones)

“Returning to the present...”



In Parallel Computing, Constants Matter

- Asymptotic analysis (big-O notation) is crucial in parallel computing, as in traditional computing

- However, constant factors also matter
  - in particular, we'll be running on a constant number of processors
    - anecdote from computational chemist colleague
  - also, since performance is a primary motivator for parallel computing, we typically want to squeeze out as many overheads as possible

Measuring Parallel Computations (Directly)

Timings: How long did the program take to run?

- typically measured in wallclock seconds (or fractions thereof)

Performance: At what rate is the program running?

- e.g., FLOPS (floating point operations per second)
  - or simply OPS
- or something more domain-specific:
  - *graph codes:* TEPS (traversed edges per second)
  - *memory bandwidth:* GB/s (gigabytes per second)
  - *table updates:* GUPS (giga-updates per second)

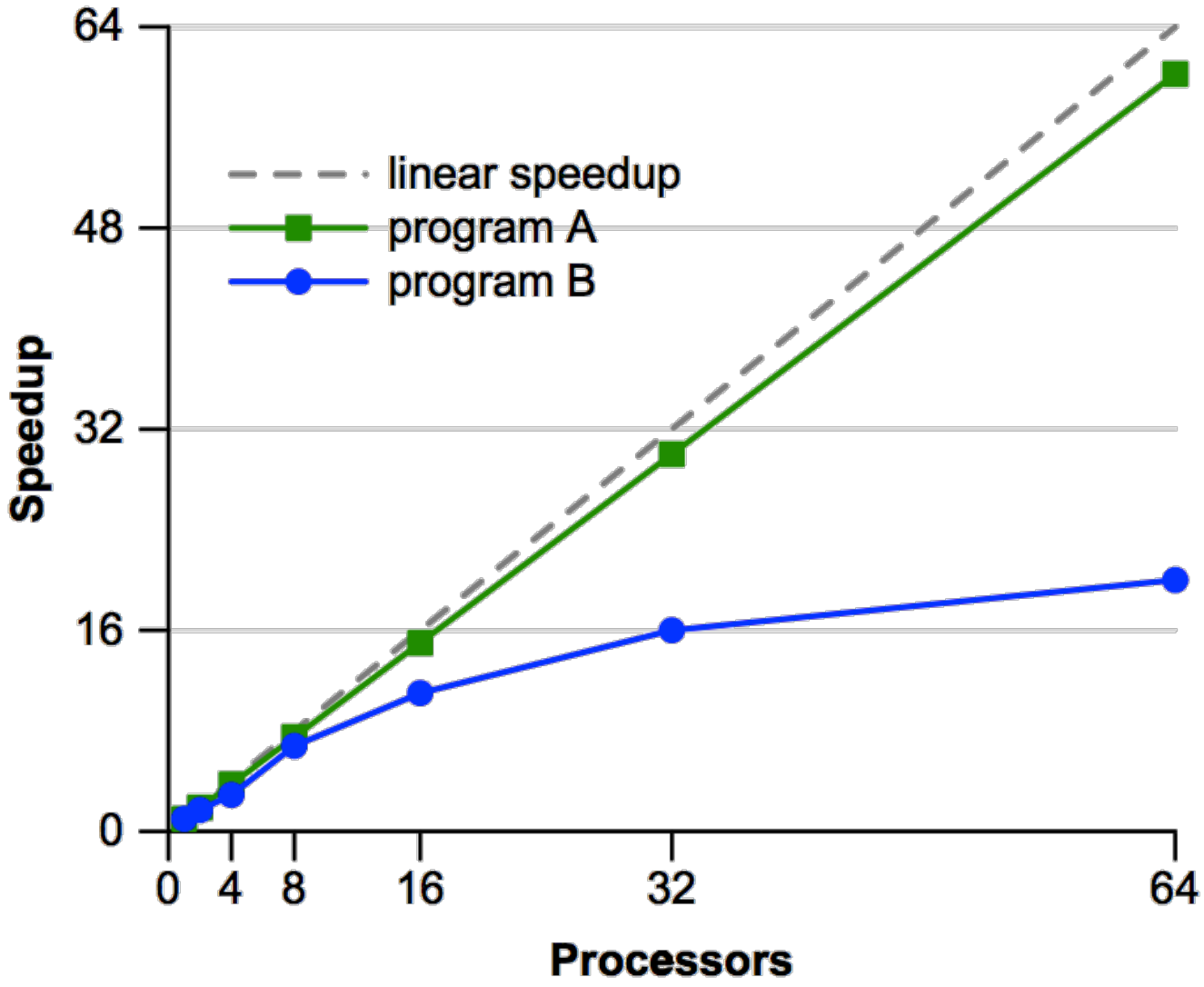
Measuring Parallel Computations (Relatively)

Speedup: How does the parallel execution compare to a serial execution?

$$Speedup_p = T_{serial} / T_p$$

Linear/Ideal Speedup: $Speedup_p = p$

Sample Speedup Graph



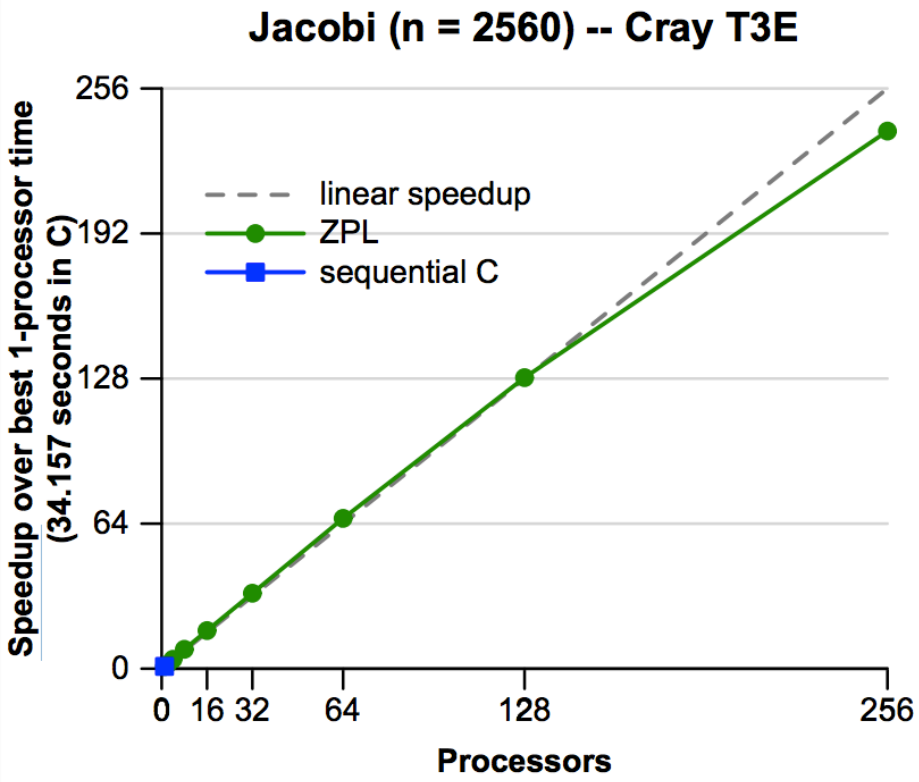
Computing Speedup: The Baseline

A key issue: What to use for the serial timing?

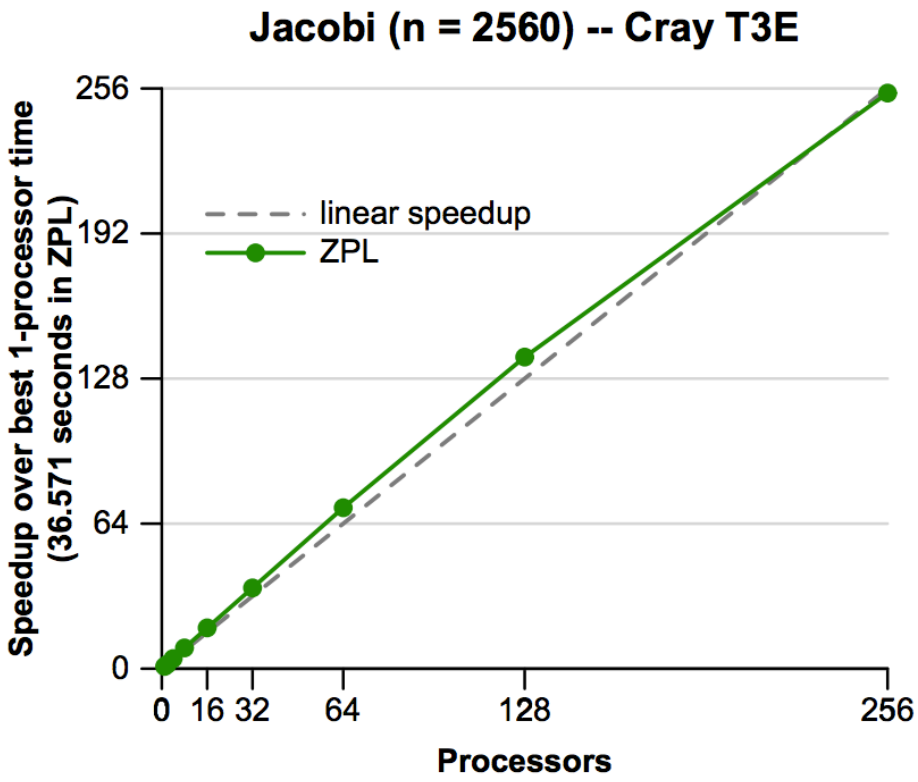
- some options:
  - the parallel code running using 1 task/processor?
  - a serial implementation of the same algorithm?
  - the best serial implementation available?
- The last is the most ideal/valuable
  - e.g., if the parallel version is 100x slower on 1 processor, computing speedup relative to itself is not very helpful
- However, depending on what's being studied, any of these choices may be reasonable

Moral: always pay attention to what speedup is computed against!

Computing Speedup relative to different baselines

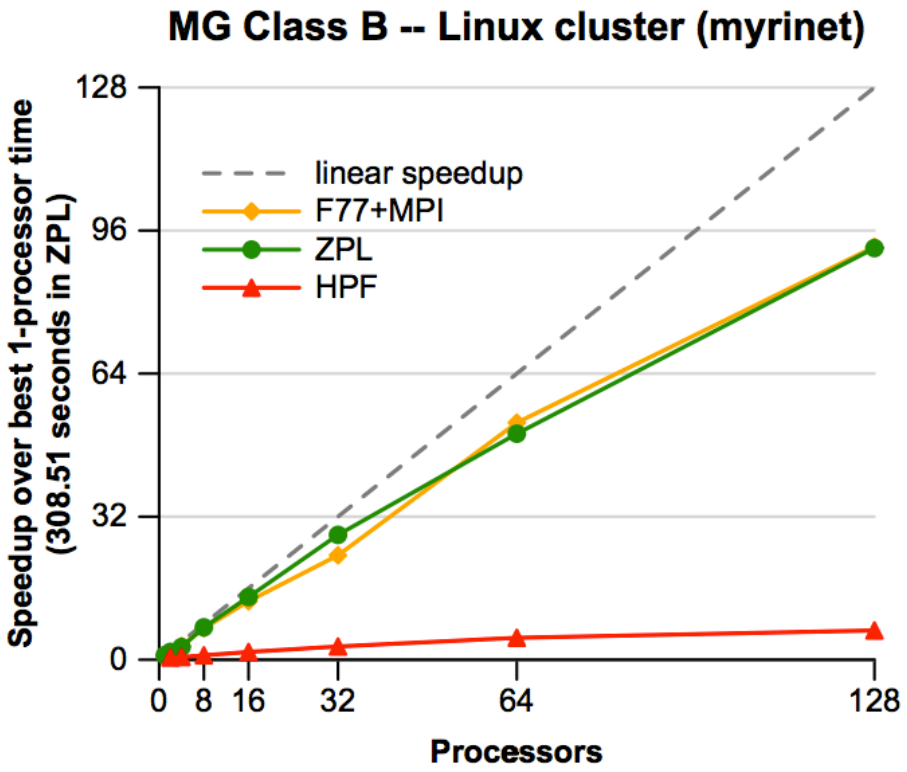


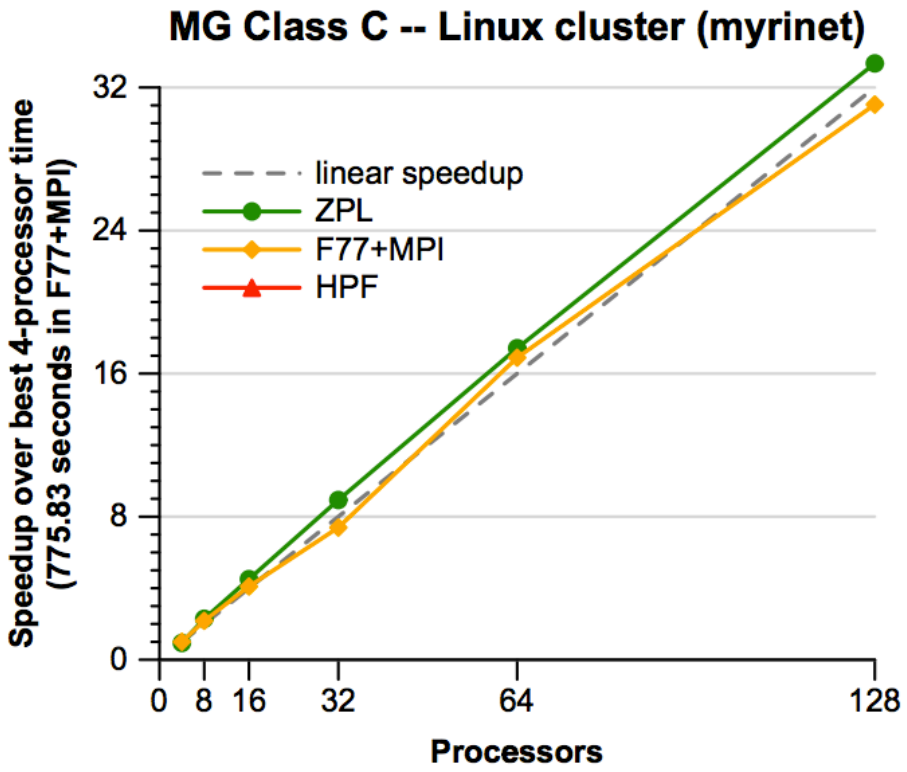
relative to a serial implementation



relative to itself

Some More Speedup Graphs





These curves are all computed using the identical serial/baseline value

Measuring Parallel Computations Relatively (2)

Efficiency: How does the parallel execution compare to a linear speedup?

$Efficiency_p = Speedup_p / p$

Ideal Efficiency: 1.0 for any value of *p*

Arguably the best way to display parallel performance

- it makes the best use of a graph's area
  - doesn't compress small scales to the lower corner
- yet it's not used in practice as much as you'd think
  - in part because it's hard to achieve 1.0, so it tends to look worse
  - also doesn't give that positive sense of "things are trending upward"

Efficiency Graph

(wouldn't a picture here be great? To the whiteboard!)

Strong vs. Weak Scalability

Strong scaling: Uses a fixed problem size as the number of processors increases.

Weak scaling: Grows the problem size with the number of processors

Both approaches have their place depending on what you care about

- again, this is something to pay attention to in interpreting parallel performance results

Embarrassingly Parallel Computations

Terminology

task: a computation that can/should execute in parallel with other tasks

- note: specific parallel programming models may attach additional characteristics to the term "task"

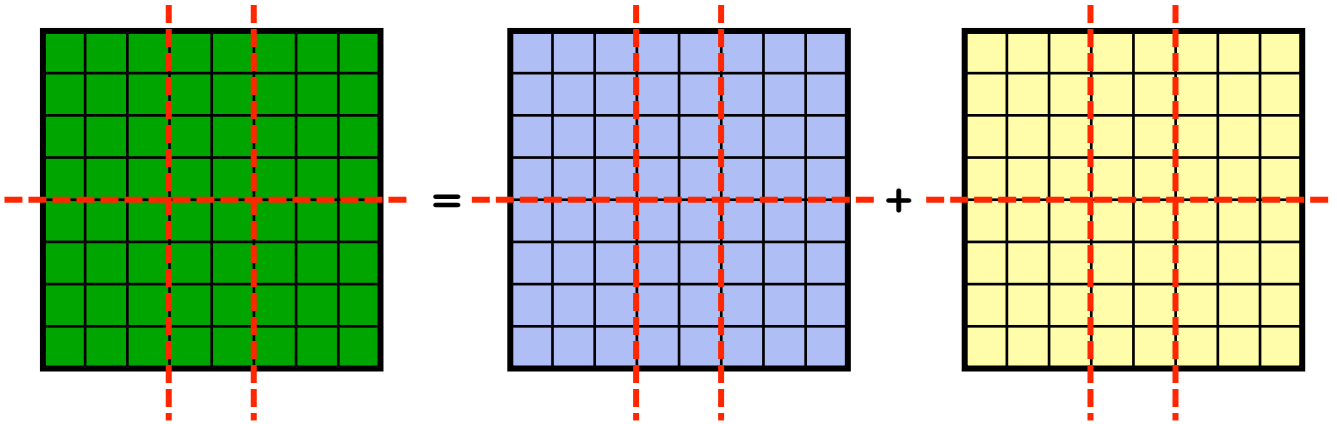
thread: a vehicle for executing tasks

- typically supplied by the HW/OS/runtime
- note that many programming models conflate the terms
  - typically when there is a 1:1 correspondence between the two

Embarrassingly Parallel

Embarrassingly Parallel Computations: Those in which the parallel tasks have no need to communicate or coordinate

- (or perhaps "no *significant* need"…)
- this is actually a very happy thing!
  - "pleasingly parallel" might therefore be a better term



Writing Embarrassingly Parallel Programs

The basic steps

1. Create the tasks
2. Have each determine what part of the problem it owns
3. Have it compute its portion of the problem
4. Wait for all the tasks to complete

Work Distribution

- Determining how to distribute the work is the main challenge
- Two common techniques:

  block: each task gets a similar-sized block of consecutive items

  cyclic: items are passed out to tasks round-robin

(wouldn't a picture here be great? To the whiteboard!)

Pthreads

Pthreads: Our first parallel programming model

Pthreads: POSIX threads

- a C-based interface for thread-based programming
- quite standard, widespread, heavily-used
- a bit painful, but valuable to have experience with
  - analogous to the value of learning assembly for C programmers
  - or learning C for performance-minded programmers

- today, we'll learn how to create tasks and wait for them to finish
- next week we'll get into more detail on coordination

Creating Pthreads

```
int pthread_create(pthread_t* tid,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(void*),
                   void* arg);
```

overall purpose: create a new thread running a task

tid: a handle to the thread, upon successful creation

attr: attributes controlling the thread's execution

- NULL => use the default attributes

start_routine: function the thread should execute

- (we might think of this as the thread's task)

arg: the argument bundle to pass to *start_routine*

returns: 0 on success, error code otherwise

Waiting for Pthreads to terminate

```
int pthread_join(pthread_t tid,
                 void** status);
```

overall purpose: wait for a specific thread to terminate/exit

- threads exit by returning from their *start_routine*
- or by calling `void pthread_exit(void* status);`

tid: the thread to wait for ("join with")

status: a handle to where its exit result should be copied

- NULL => I don't care about the status

returns: 0 on success, error code otherwise

Trivial Pthread Create/Join Example

```
int resultLoc;

void* addone(void* arg) {
  int argAsInt = *(int*)arg;
  printf("thread running foo(%d)\n", argAsInt);
  resultLoc = argAsInt + 1;
  return &resultLoc;
}

pthread_t myThread;
int arg = 3;
pthread_create(&myThread, NULL, addone, &arg);

void* result;
pthread_join(myThread, &result);

printf("Original thread got %d as result\n", *(int*)result);
```

Full Pthread Create/Join Example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int resultLoc;

void* addone(void* arg) {
  int argAsInt = *(int*)arg;
  printf("thread running foo(%d)\n", argAsInt);
  resultLoc = argAsInt + 1;
  return &resultLoc;
}

int main(int argc, char* argv[]) {
  pthread_t myThread;

  int arg = 3;
  int err;
  if (err = pthread_create(&myThread, NULL,
                           addone, &arg)) {
    printf("pthread_create() got an error!\n");
    exit(1);
  }

  void* result;
  if (err = pthread_join(myThread, &result)) {
    printf("pthread_join() got an error!\n");
    exit(1);
  }

  printf("Original thread got %d as result\n", *(int*)result);
}
```

This week’s assignment

Goal

- Get everyone's feet wet and ready to go

- Items:
  - Course Survey
  - Reading
    - 3 items this week:
      - "The Free Lunch is Over"
      - selections from Lin & Snyder
      - Chapel intro
    - need to send in a few discussion questions by Monday evening
  - SW Installation: Fedora 17 VM, Chapel
  - Embarrassingly Parallel Performance Study

Embarrassingly Parallel (EP) Study

- You're going to write and study two programs:
  - 1 in C+Pthreads, 1 in Chapel
  - input parameters: problem size, # tasks
  - each will allocate array, initialize it, do EP computation
- Study this space:
  - *computations:* negate array element vs. compute factorial
  - *input deck:* random values vs. value ramp
  - *distributions:* block vs. cyclic
  - *numbers of tasks:* 1..#cores
- Predict which will perform best, measure, analyze

That's it for today!

- Questions?