A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency

Lecture 1
Introduction to Multithreading & Fork-Join Parallelism

Steve Wolfman, based on work by Dan Grossman

(with tiny tweaks by Alan Hu)

Learning Goals

By the end of this unit, you should be able to:

•Distinguish between parallelism—improving performance by exploiting multiple processors—and concurrency—managing simultaneous access to shared resources.

•Explain and justify the task-based (vs. thread-based) approach to parallelism. (Include asymptotic analysis of the approach and its practical considerations, like "bottoming out" at a reasonable level.)

•Define “map” and “reduce”, and explain how they can be useful.

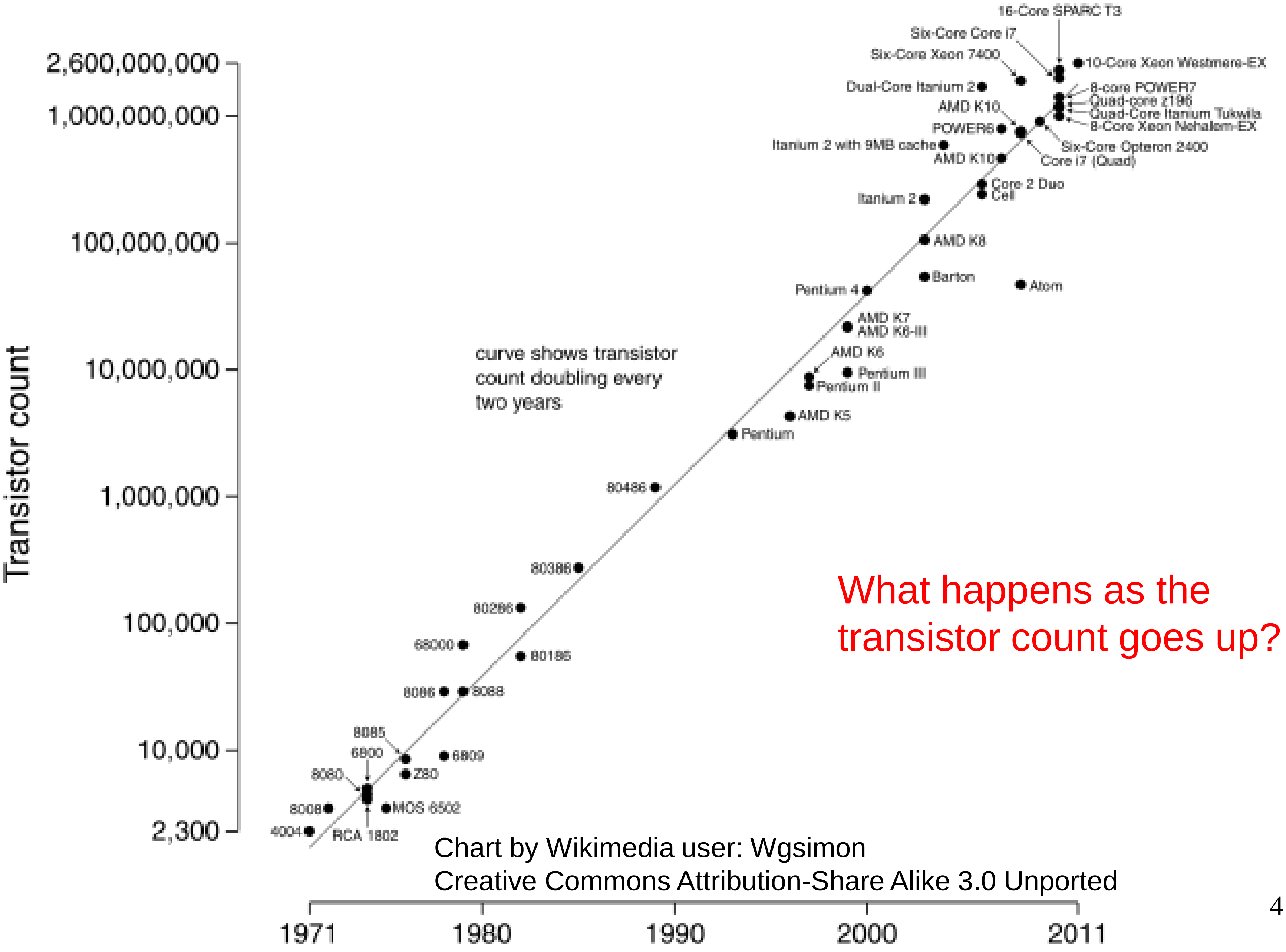
•Define work, span, speedup, and Amdahl’s Law.

•Write simple fork-join and divide-and-conquer programs in C++11 and with OpenMP.

Outline

- History and Motivation
- Parallelism and Concurrency Intro
- Counting Matches
    - Parallelizing
    - Better, more general parallelizing

Microprocessor Transistor Counts 1971-2011 & Moore's Law



What happens as the transistor count goes up?

Chart by Wikimedia user: Wgsimon
Creative Commons Attribution-Share Alike 3.0 Unported

(zoomed in)

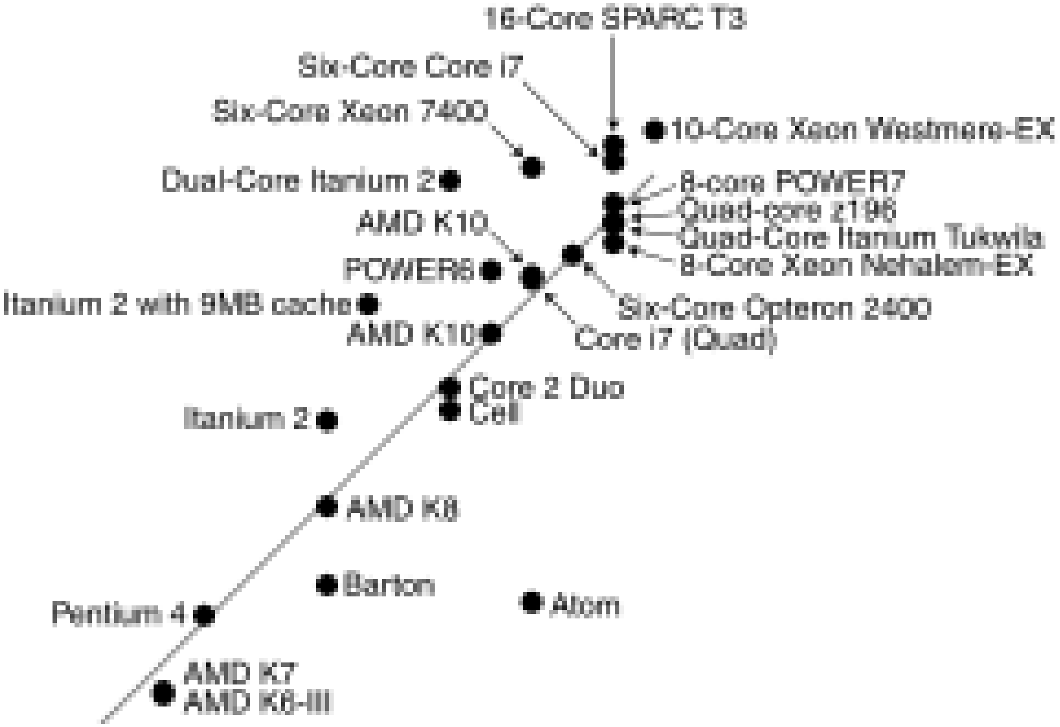


Chart by Wikimedia user: Wgsimon
Creative Commons Attribution-Share Alike 3.0 Unported

(Goodbye to) Sequential Programming

One thing happens at a time.

The next thing to happen is “my” next instruction.

Removing this assumption creates major challenges & opportunities

- Programming: Divide work among threads of execution and coordinate (synchronize) among them
- Algorithms: How can parallel activity provide speed-up? (more throughput: work done per unit time)
- Data structures: May need to support concurrent access (multiple threads operating on data at the same time)

A simplified view of history

Writing multi-threaded code in common languages like Java and C is more difficult than single-threaded (sequential) code.

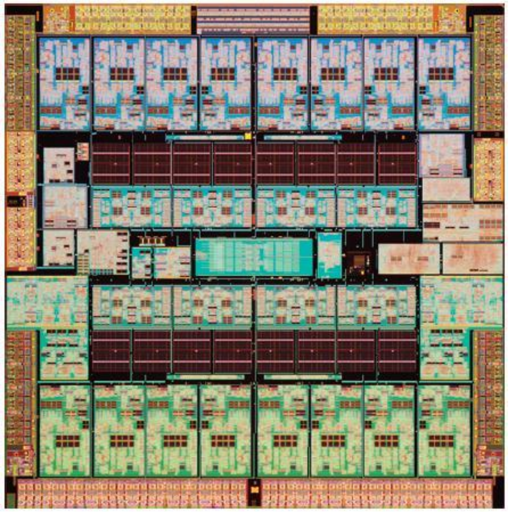
So, as long as possible (~1980-2005), desktop computers' speed running sequential programs doubled every ~2 years.

Although we keep making transistors/wires smaller, we don't know how to continue the speed increases:

- Increasing clock rate generates too much heat
- Relative cost of memory access is too high

Solution, not faster but smaller and *more*…

(Sparc T3 micrograph from Oracle; 16 cores.)



A simpli

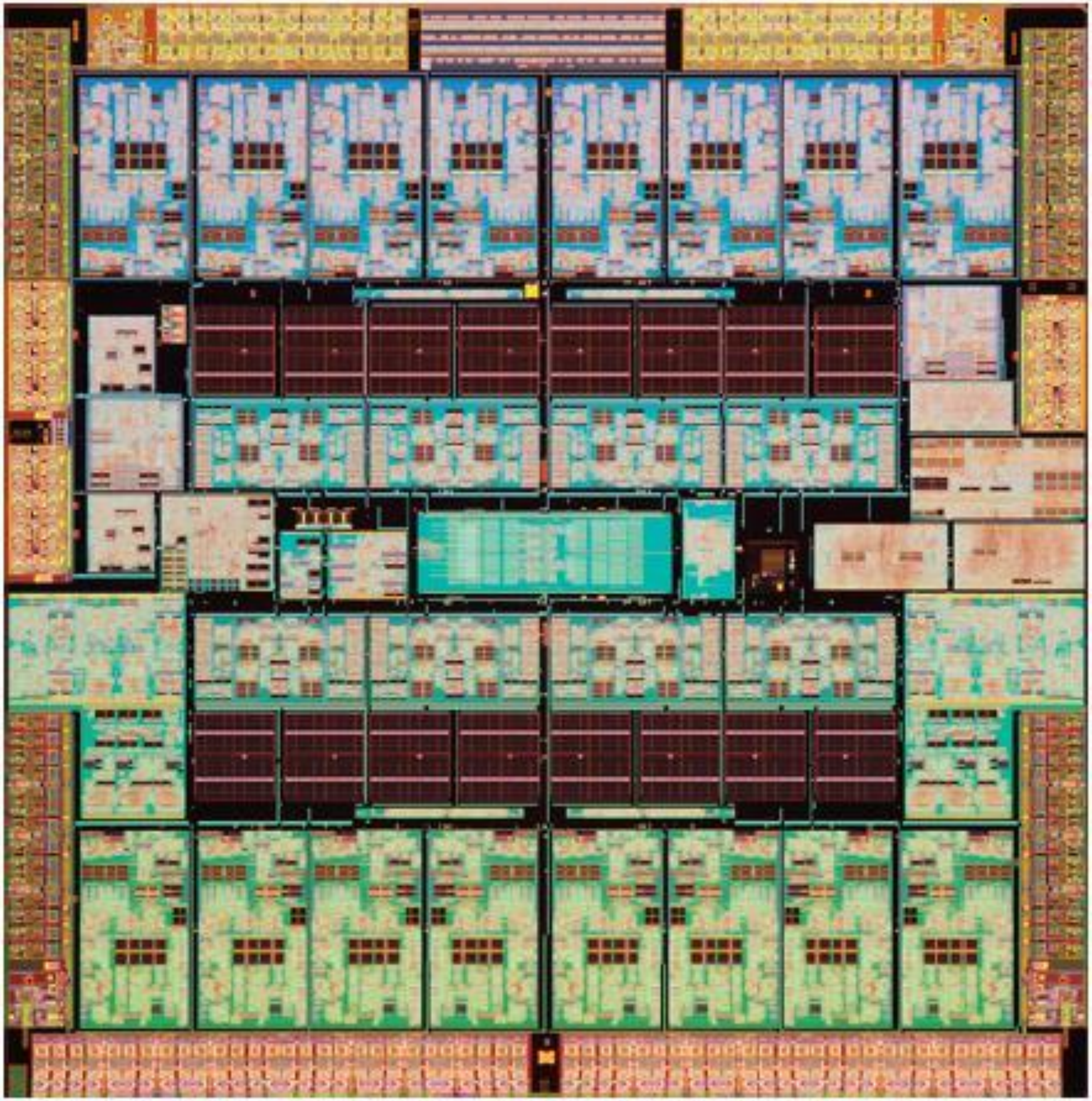
Writing multi
is more d

So, as long a
running s

Although we
we don't l

- Increa
- Relati

Solution, not



What to do with multiple processors?

- Run multiple totally different programs at the same time

  (Already doing that, but with time-slicing.)

- **Do multiple things at once in one program**
  - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations

Outline

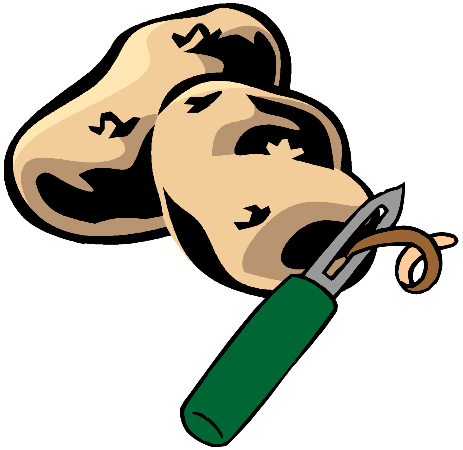
- History and Motivation
- Parallelism and Concurrency Intro
- Counting Matches
  - Parallelizing
  - Better, more general parallelizing

KP Duty: Peeling Potatoes, Parallelism

How long does it take a person to peel one potato? Say: 15s

How long does it take a person to peel 10,000 potatoes?
~2500 min = ~42hrs = ~one week full-time.

How long would it take 100 people with 100 potato peelers to peel 10,000 potatoes?



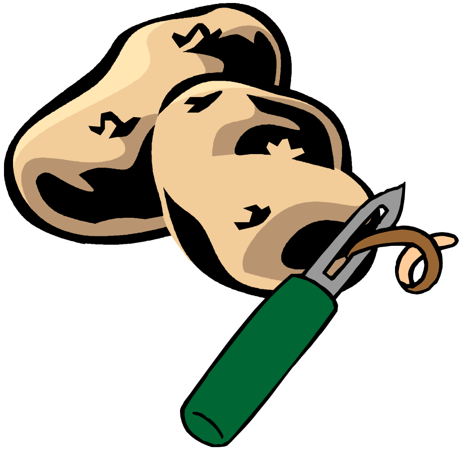
KP Duty: Peeling Potatoes, Parallelism

How long does it take a person to peel one potato? Say: 15s

How long does it take a person to peel 10,000 potatoes?
~2500 min = ~42hrs = ~one week full-time.

How long would it take 100 people with 100 potato peelers to peel 10,000 potatoes?

Parallelism: using extra resources to solve a problem faster.



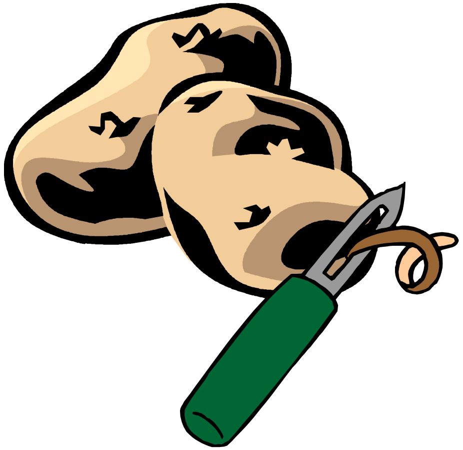
Note: these definitions of "parallelism" and "concurrency" are not yet standard but the perspective is essential to avoid confusion!

KP Duty: Peeling Potatoes, Concurrency

How long does it take a person to peel one potato? Say: 15s

How long does it take a person to peel 10,000 potatoes?
~2500 min = ~42hrs = ~one week full-time.

How long would it take 2 people with 1 potato peeler to peel 10,000 potatoes?



KP Duty: Peeling Potatoes, Concurrency

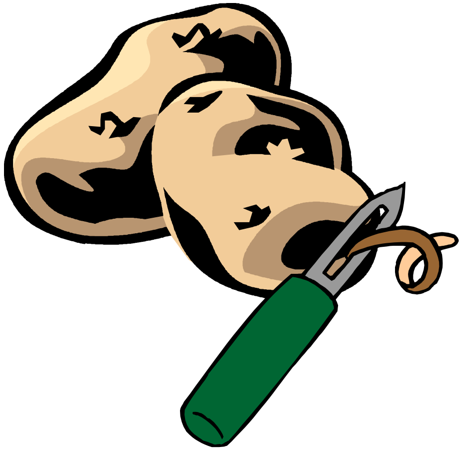
How long does it take a person to peel one potato? Say: 15s

How long does it take a person to peel 10,000 potatoes?
~2500 min = ~42hrs = ~one week full-time.

How long would it take 2 people with 1 potato peeler to peel 10,000 potatoes?

Concurrency: Correctly and efficiently manage access to shared resources

(Better example: Lots of cooks in one kitchen, but only 4 stove burners. Want to allow access to all 4 burners, but not cause spills or incorrect burner settings.)



Note: these definitions of "parallelism" and "concurrency" are not yet standard but the perspective is essential to avoid confusion!

Models of Computation

- When you first learned to program in a sequential language like Java, C, C++, etc., you had an abstract model of a computer:
  - CPU processes data
  - Memory stores data

Models of Computation

- When you first learned to program in a sequential language like Java, C, C++, etc., you had an abstract model of a computer:
  - CPU processes data
    - Fetch-Decode-Execute Cycle: Grab instructions one at a time, and do them.
    - Program Counter: Keep track of where you are in the code.
  - Memory stores data
    - Local Variables
    - Global Variables
    - Heap-Allocated Objects

Models of Computation

- When you first learned to program in a sequential language like Java, C, C++, etc., you had an abstract model of a computer:
  - CPU processes data
    - Fetch-Decode-Execute Cycle: Grab instructions one at a time, and do them.
    - Program Counter: Keep track of where you are in the code. **(Also a call stack to track of function calls.)**
  - Memory stores data
    - Local Variables **(Stored in stack frame on call stack).**
    - Global Variables
    - Heap-Allocated Objects

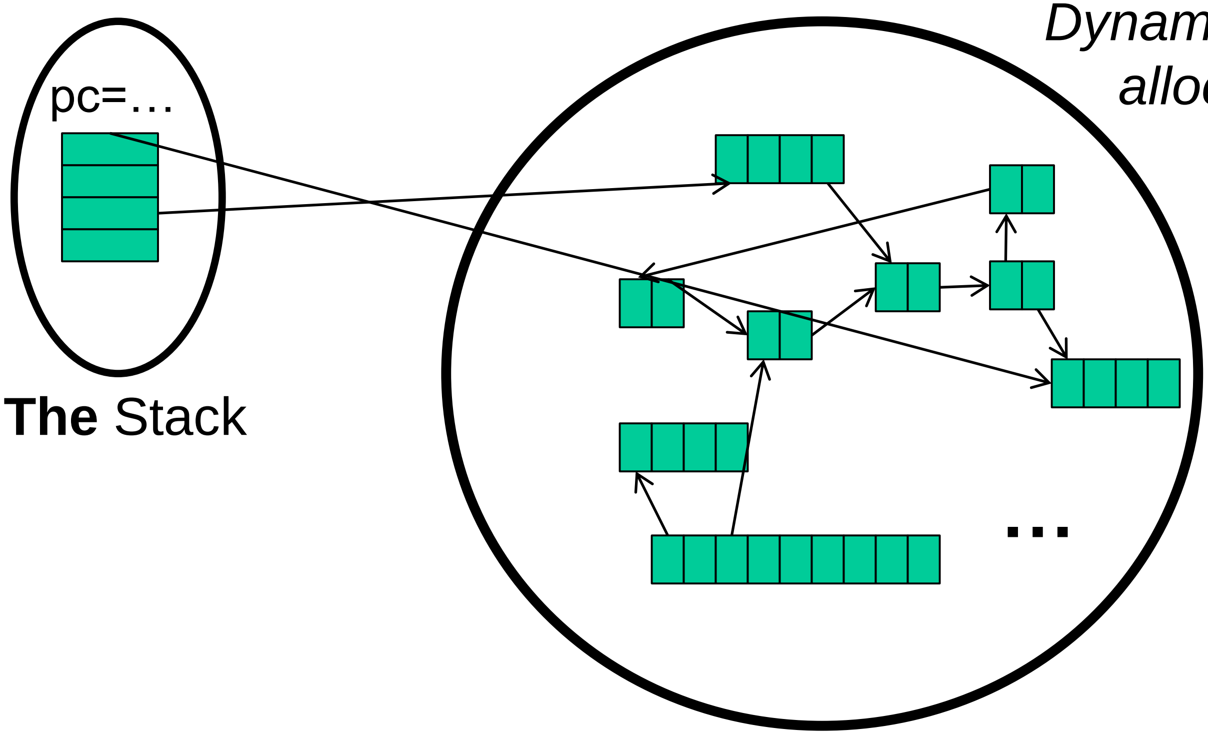
Models of Parallel Computation

- **There are many different ways to model parallel computation, which model which of these are shared or distinct…**
  - CPU processes data
    - Fetch-Decode-Execute Cycle: Grab instructions one at a time, and do them.
    - Program Counter: Keep track of where you are in the code. **(Also a call stack to track of function calls.)**
  - Memory stores data
    - Local Variables **(Stored in stack frame on call stack).**
    - Global Variables
    - Heap-Allocated Objects

Models of Parallel Computation

- In this course, we will work with the **shared memory** model of parallel computation.
  - This is currently the most widely used model.
    - Communicate by reading/writing variables – nothing special needed.
    - Therefore, fast, lightweight communication
    - Close to how hardware behaves on small multiprocessors
  - However, there are good reasons why many people argue that this isn’t a good model over the long term:
    - Easy to make subtle mistakes
    - Not how hardware behaves on big multiprocessors – memory isn’t truly shared.

OLD *Memory Model*



Local variables
Control flow info

The Stack

Dynamically allocated data.

The Heap

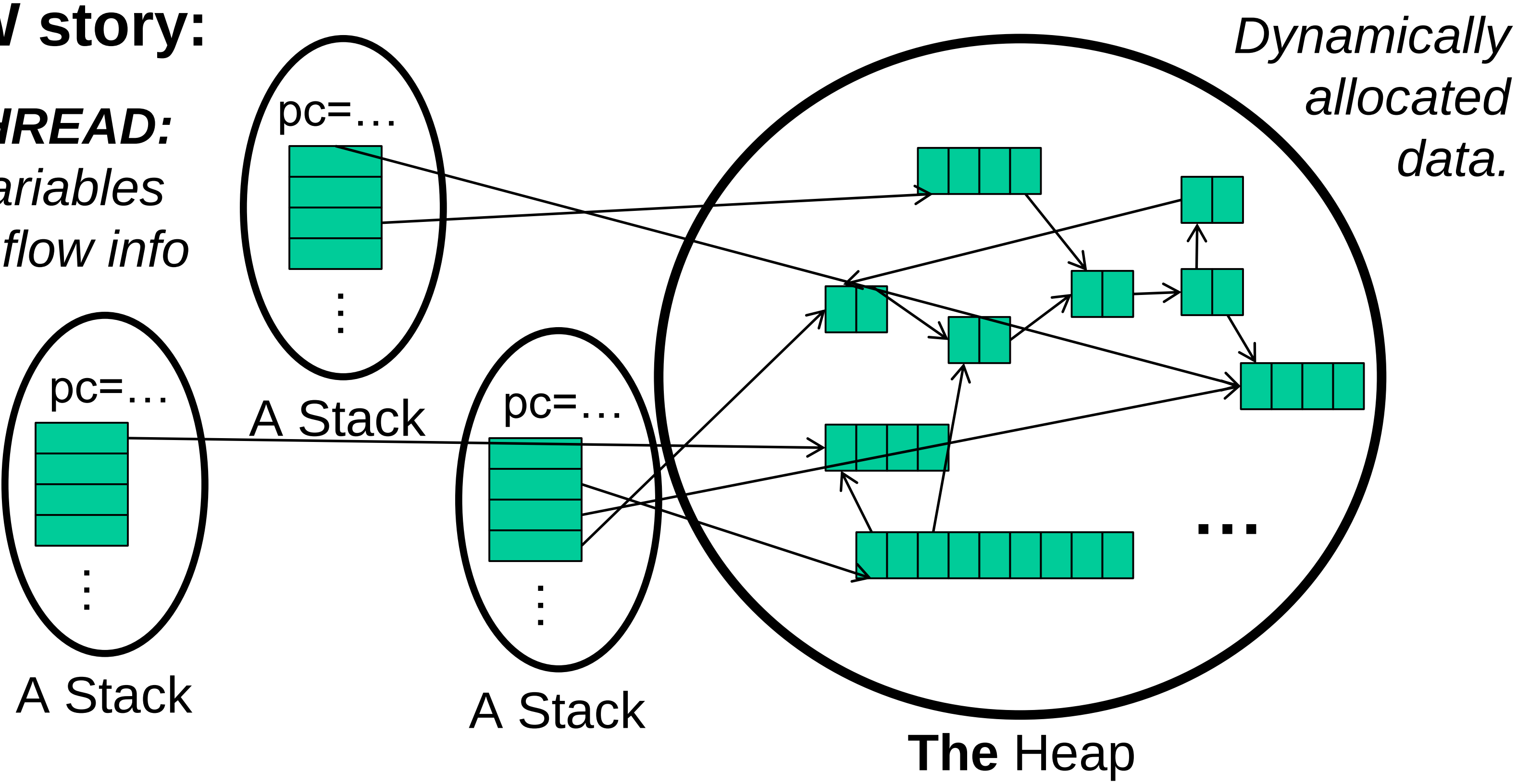
(pc = program counter, address of current instruction)

Shared Memory Model

We assume (and C++11 specifies) shared memory w/explicit threads

NEW story:

PER THREAD:
Local variables
Control flow info

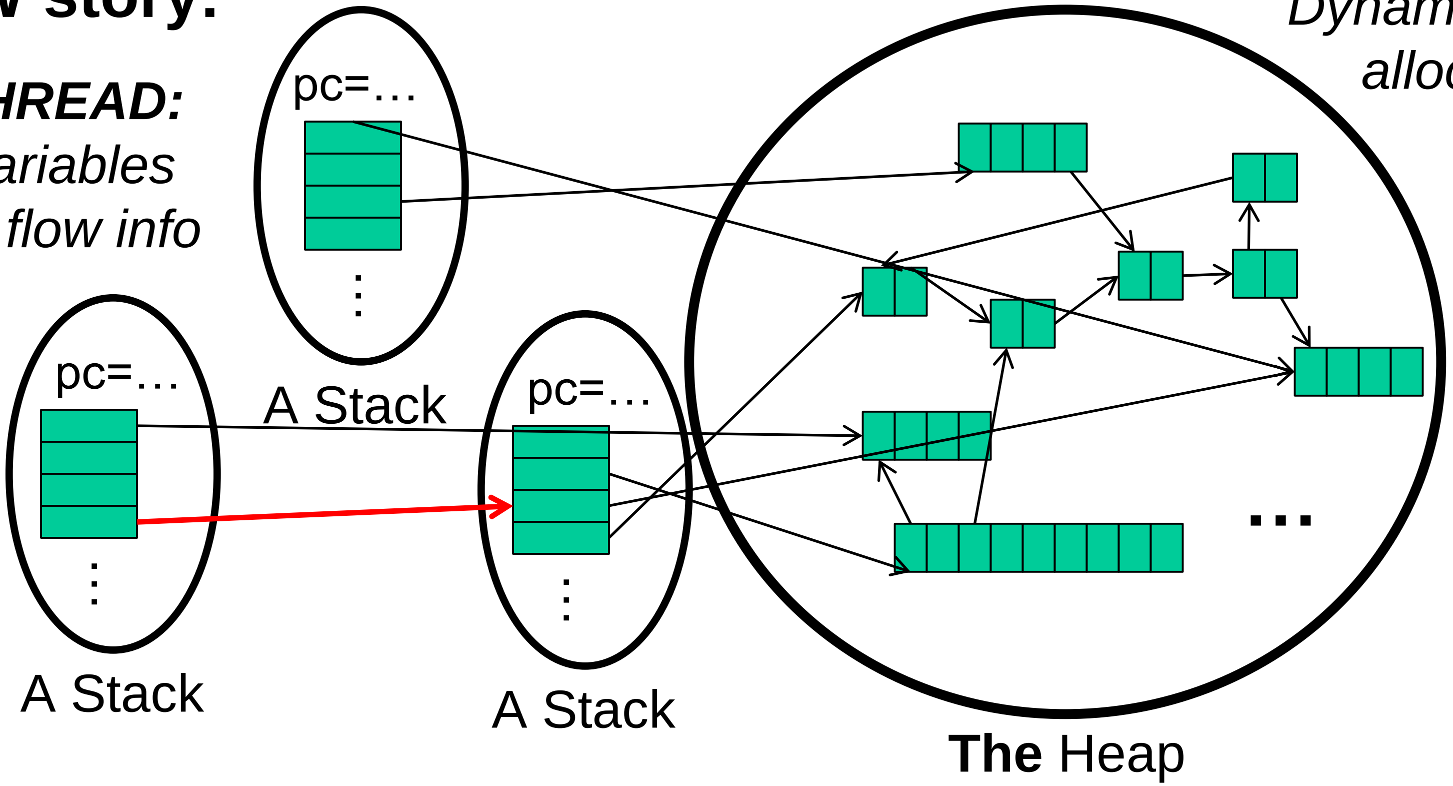


Shared Memory Model

We assume (and C++11 specifies) shared memory w/explicit threads

NEW story:

PER THREAD:
Local variables
Control flow info



Dynamically allocated data.

Note: we can share ***local*** variables by sharing pointers to their locations.

Other models

We will focus on shared memory, but you should know several other models exist and have their own advantages

- Message-passing: Each thread has its own collection of objects. Communication is via explicitly sending/receiving messages
  - Cooks working in separate kitchens, mail around ingredients
- Dataflow: Programmers write programs in terms of a DAG.

  A node executes after all of its predecessors in the graph
  - Cooks wait to be handed results of previous steps
- Data parallelism: Have primitives for things like “apply function to every element of an array in parallel”

Outline

- History and Motivation
- Parallelism and Concurrency Intro
- Counting Matches
  - Parallelizing
  - Better, more general parallelizing

Problem: Count Matches of a Target

- How many times does the number 3 appear?

3	5	9	3	2	0	4	6	1	3

```
// Basic sequential version.
int count_matches(int array[], int len, int target) {
  int matches = 0;
  for (int i = 0; i < len; i++) {
    if (array[i] == target)
      matches++;
  }
  return matches;
}
```

How can we take advantage of parallelism?

First attempt (wrong.. but grab the code!)

```
void cmp_helper(int * result, int array[],
                int lo, int hi, int target) {
  *result = count_matches(array + lo, hi - lo, target);
}

int cm_parallel(int array[], int len, int target) {
  int divs = 8;

  std::thread workers[divs];
  int results[divs];
  for (int d = 0; d < div; d++)
    workers[d] = std::thread(&cmp_helper,
      &results[d], array, (d*len)/divs,
      ((d+1)*len)/divs, target);

  int matches = 0;
  for (int d = 0; d < divs; d++)
    matches += results[d];

  return matches;
}
```

Notice: we use a pointer to shared memory to communicate across threads!

Shared memory?

Beware sharing memory like the pointer to an element of the **`matchesPer`** array!

- Race condition: What happens if multiple threads try to write it at once (or one tries to write while others read)?
  KABOOM (possibly silently!)
- Scope problems: What happens if the child thread is still using the variable when it is deallocated (goes out of scope) in the parent? KABOOM (possibly silently!)

So… what's C++'s problem, and why did it give us an error?

Join (not the most descriptive word)

- The `thread` class defines various methods you could not implement on your own
  - For example, the constructor calls its argument *in a new thread*
- The `join` method helps coordinate this kind of computation
  - Caller blocks until/unless the receiver is done executing (i.e., its constructor's argument function returns)
  - Else we have a race condition accessing `matchesPer[d]`
- This style of parallel programming is called "fork/join"

That should kill two birds with one stone.

Fix the code and do some timings!

First attempt (patched!)

```
int cm_parallel(int array[], int len, int target) {
  int divs = 8;

  std::thread workers[divs];
  int results[divs];
  for (int d = 0; d < div; d++)
    workers[d] = std::thread(&cmp_helper, &results[d],
      array, (d*len)/divs, ((d+1)*len)/divs,
      target);

  int matches = 0;
  for (int d = 0; d < divs; d++) {
    workers[d].join();
    matches += results[d];
  }

  return matches;
}
```

Outline

- History and Motivation
- Parallelism and Concurrency Intro
- Counting Matches
  - Parallelizing
  - Better, more general parallelizing

Success! Are we done?

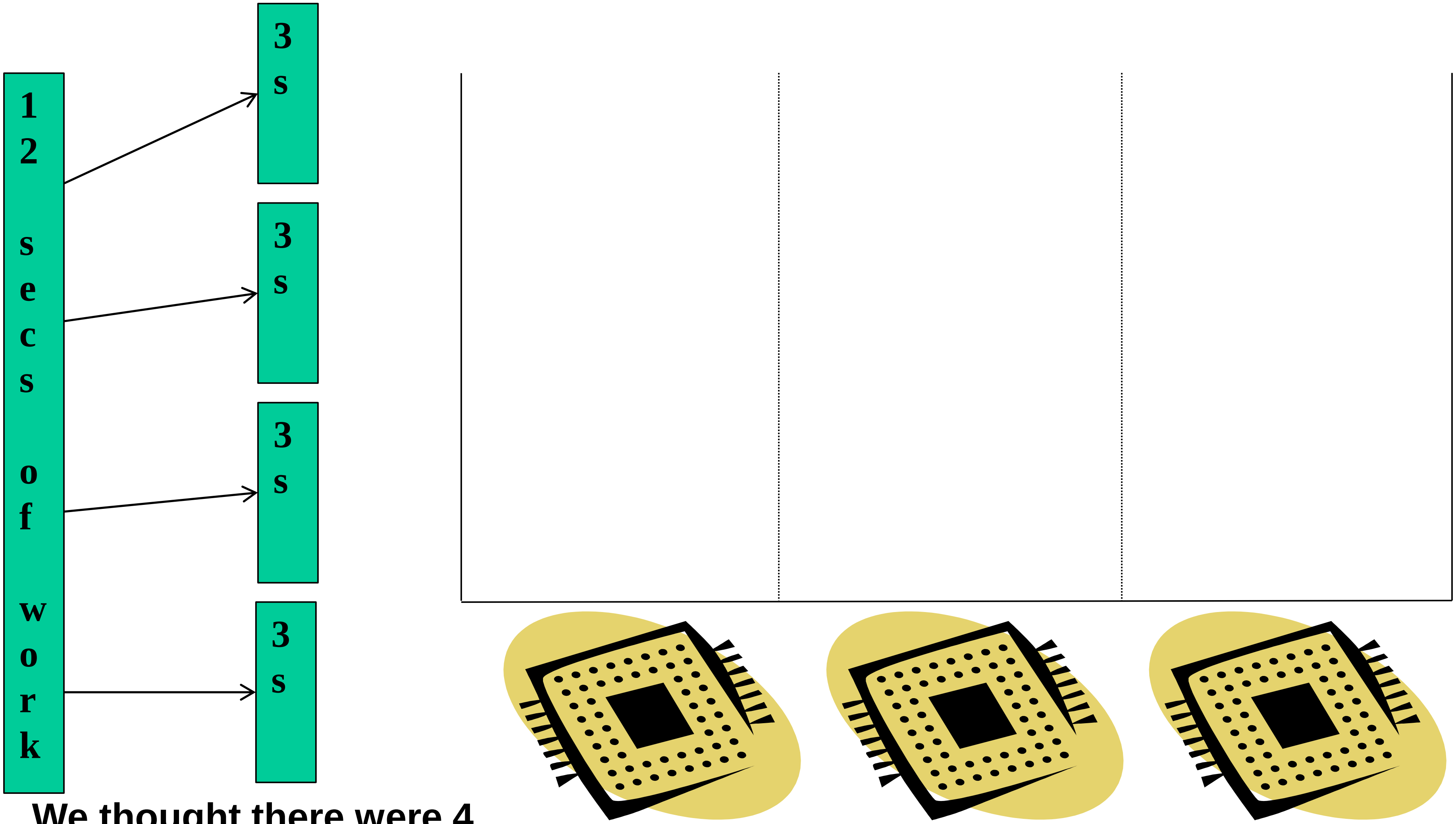
Answer these:

- What happens if I run my code on an old-fashioned one-core machine?

- What happens if I run my code on a machine with *more* cores in the future?

(Done? Think about how to fix it and do so in the code.)

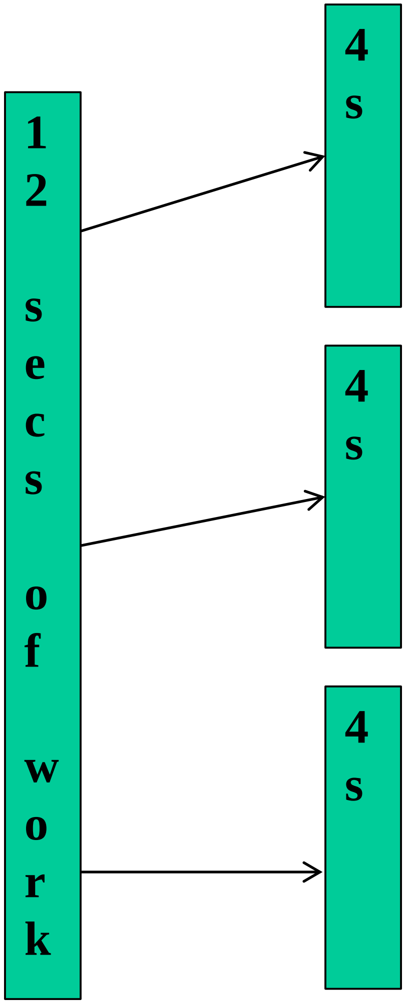
Chopping (a Bit) Too Fine



We thought there were 4 processors available.

But there's only 3. Result?

Chopping Just Right

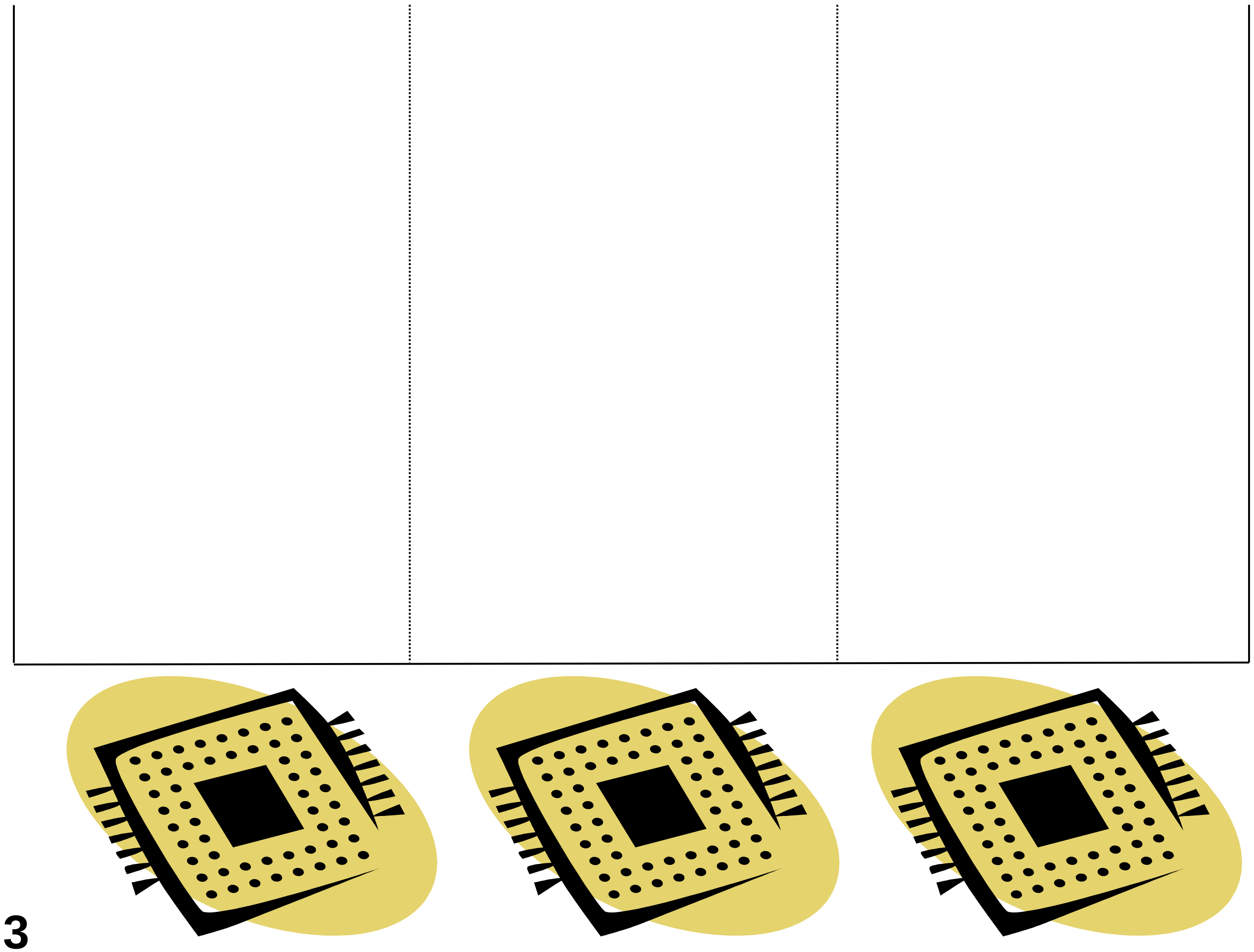


12 secs of work

4 s

4 s

4 s



We thought there were 3 processors available.

And there are. Result?

Success! Are we done?

Answer these:

- What happens if I run my code on an old-fashioned one-core machine?

- What happens if I run my code on a machine with *more* cores in the future?

- Let’s *fix* these!

(Note: std::thread::hardware_concurrency() and omp_get_num_procs().)

Success! Are we done?

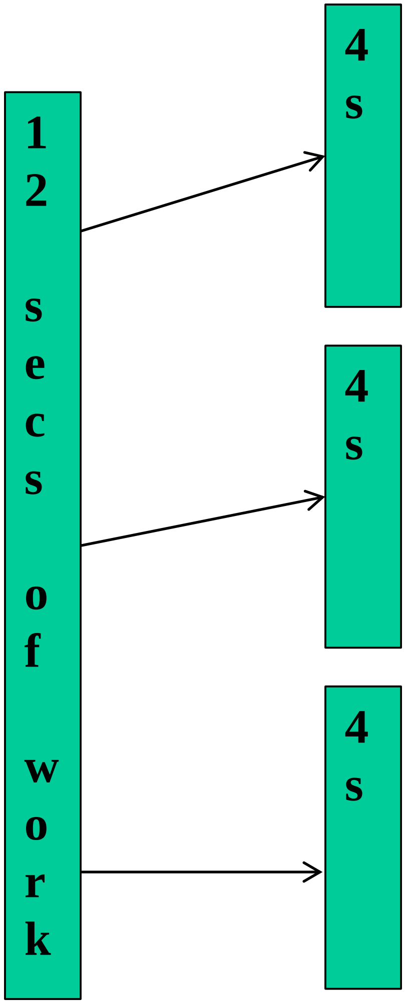
Answer these:

- Might your prof somehow get better parallel performance than you? Why? (Note: your prof has arranged for a machine that no one else can log into. Nyah, nyah!)

- Might your performance vary as the whole class tries problems, depending on when you start your run?

(Done? Think about how to fix it and do so in the code.)

Is there a “Just Right”?

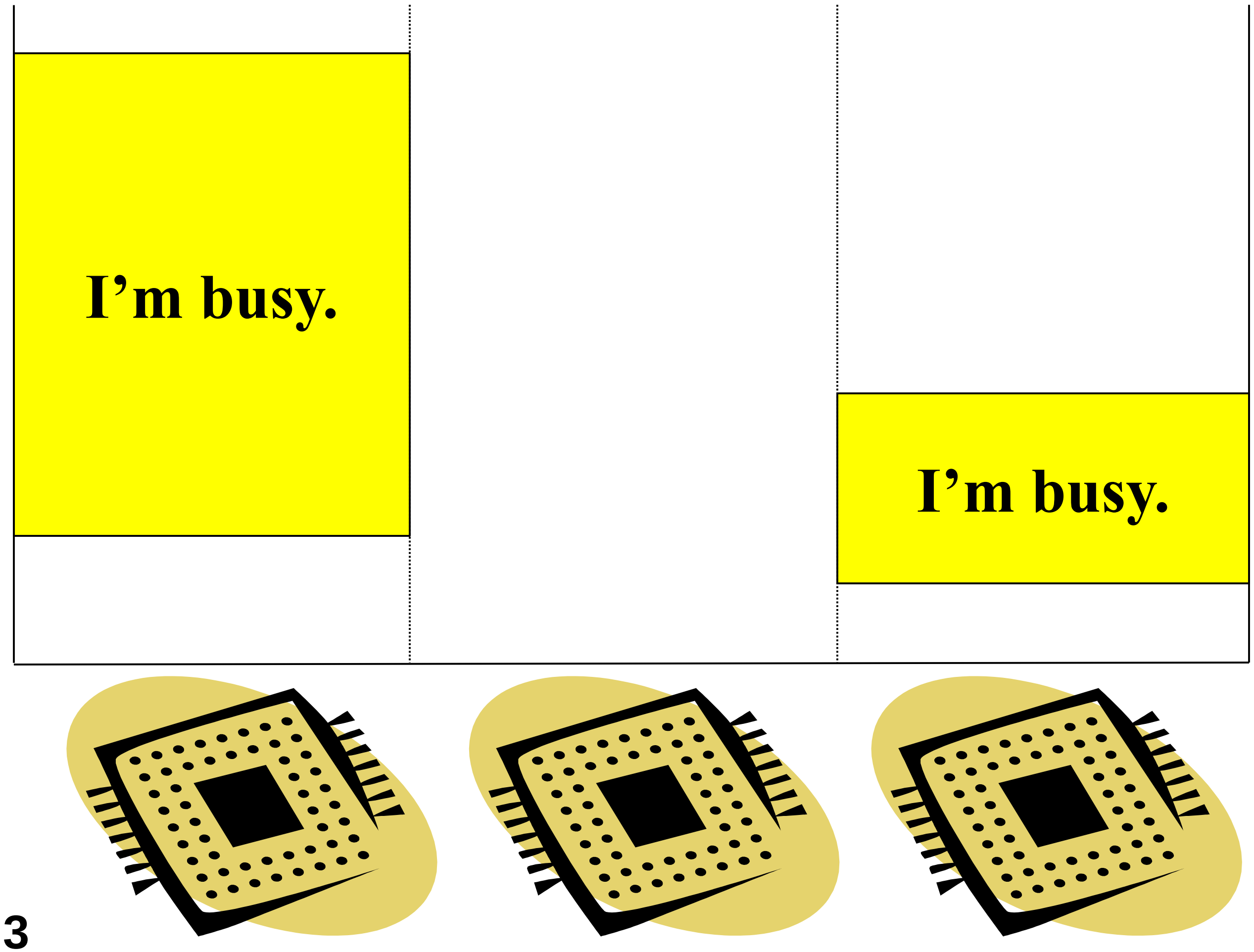


12 secs of work

4 s

4 s

4 s



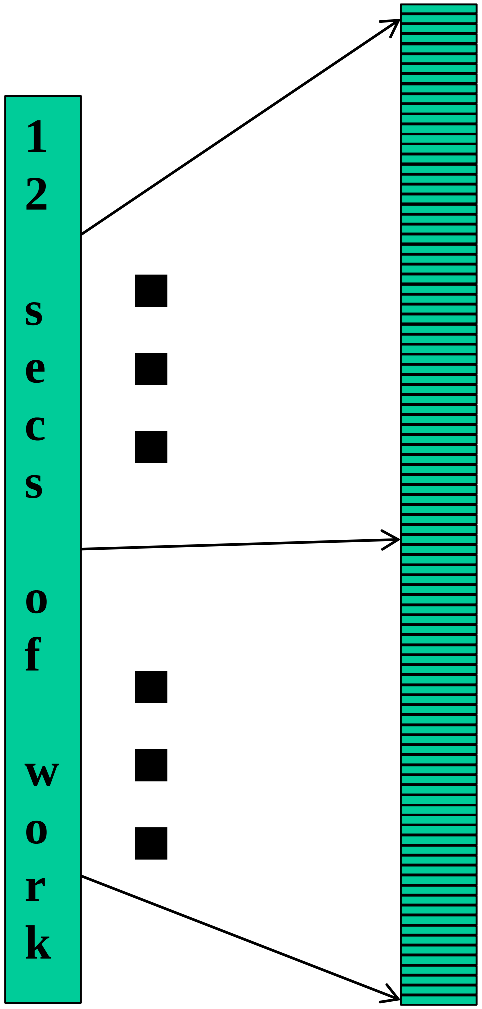
I’m busy.

I’m busy.

We thought there were 3 processors available.

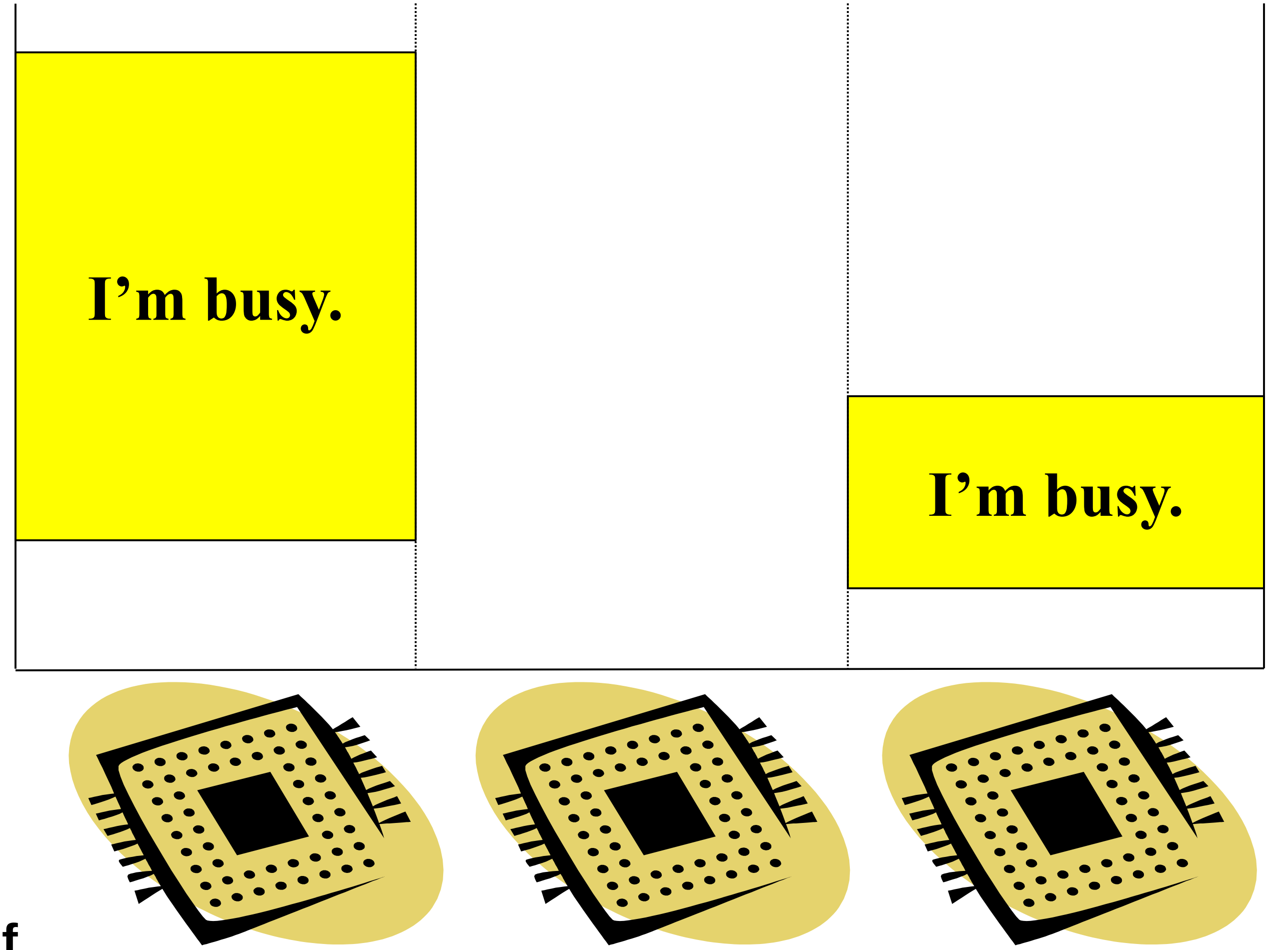
And there are. Result?

Chopping So Fine It’s Like Sand or Water



12 secs of work

(of course, we can’t predict the busy times!)



I’m busy.

I’m busy.

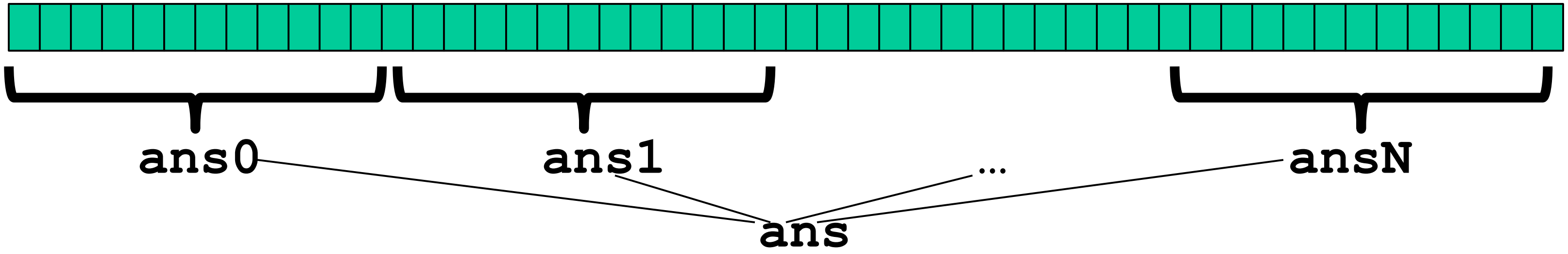
We chopped into lots of pieces.

And there are a few processors. Result?

A Better Approach

Counterintuitive solution: use far more threads than # of processors

- For constant-factor reasons, we will abandon C++'s threads. From here on out, we call these "tasks" instead b/c they're assignable to threads but not necessarily threads themselves.



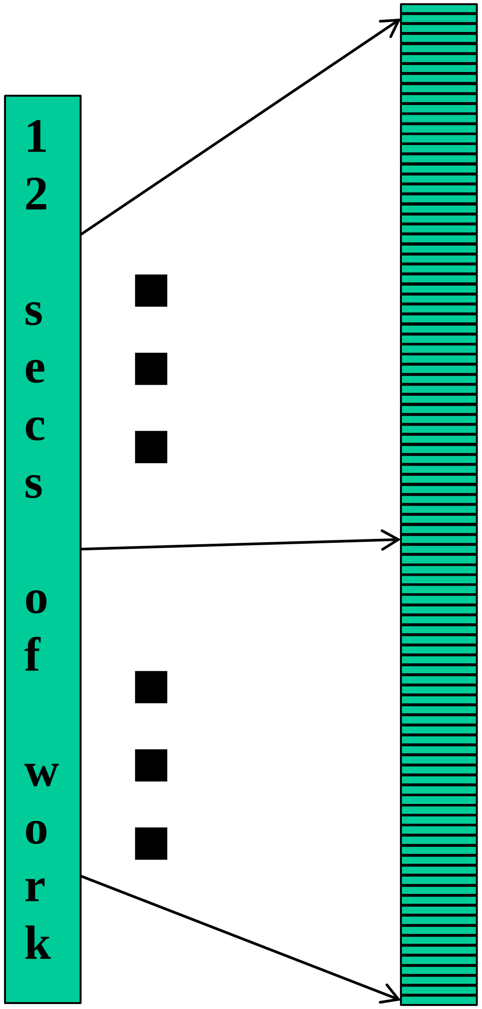
1. Forward-portable: Lots of helpers each doing a small task.
2. Processors available: Hand out tasks as you go
   - If 3 processors available and have 100 tasks, then ignoring constant-factor overheads, extra time is < 3%
3. Load imbalance: If one task actually takes much more time? No problem if scheduled early enough, and variation (factor of 10x?) probably small if tasks are small

Success! Are we done?

Answer these:

- Might your prof somehow get better parallel performance than you? Why? (Note: your prof has arranged for a machine that no one else can log into. Nyah, nyah!)
- Might your performance vary as the whole class tries problems, depending on your typing speed?
- Let's *fix* these!

Chopping Too Fine Again



We chopped into n pieces (n == array length).

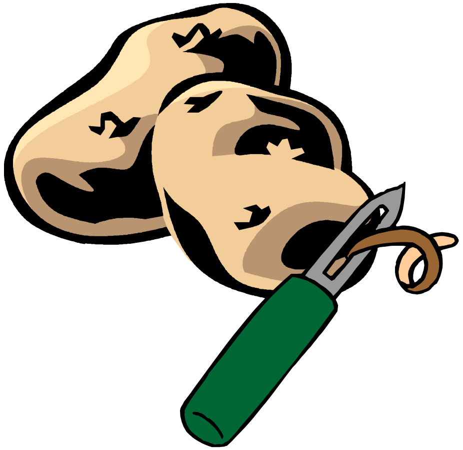
Result?

KP Duty: Peeling Potatoes, *Parallelism Remainder*

How long does it take a person to peel one potato? Say: 15s

How long does it take a person to peel 10,000 potatoes?
~2500 min = ~42hrs = ~one week full-time.

How long would it take 100 people with 100 potato peelers to peel 10,000 potatoes?



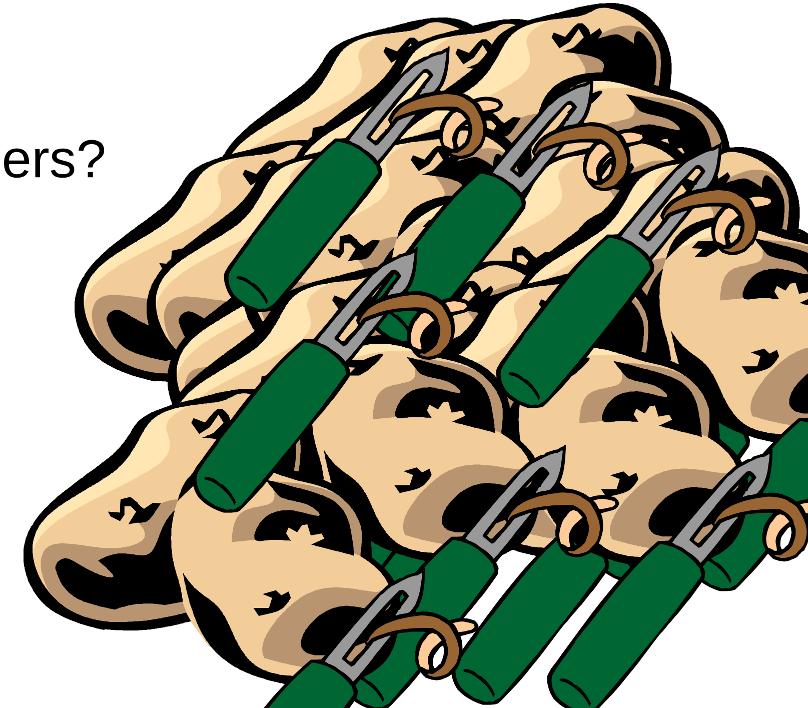
KP Duty: Peeling Potatoes, Parallelism Problem

How long does it take a person to peel one potato? Say: 15s

How long does it take a person to peel 10,000 potatoes?
~2500 min = ~42hrs = ~one week full-time.

How long would it take 10,000 people with 10,000 potato peelers to peel 10,000 potatoes?

How about 5,000 people with 5,000 peelers?



OpenMP Library

- Even with all this care, C++11's threads are usually too "heavyweight" (implementation dependent).
- OpenMP is a standard library that provides very lightweight threads, so we can chop tasks up very finely.
- We will see OpenMP code soon, but first, we want to see a better way to divide up a job into smaller tasks…

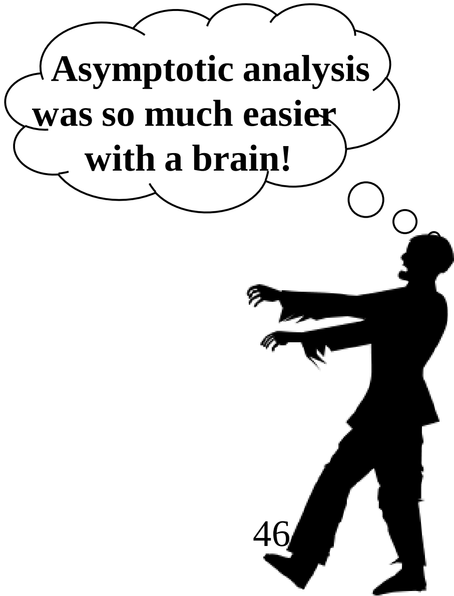
How *Do We Infect the Living World?*

Problem: A group of (non-Computer Scientist) zombies asks for your help infecting the living. Each time a zombie bites a human, it also gets to transfer a program.

Currently, the new zombie in town has the humans line up and proceeds from one to the next, biting and transferring the null program (do nothing, except say “Eat Brains!!”).

Analysis?

How do they do better?



How *Do We Divide Up the Work?*

The metaphor is not perfect. Each time we “infect” a processor, it goes off and does useful work. However, the analysis still holds:

Let n be the array size and P be the number of processors.

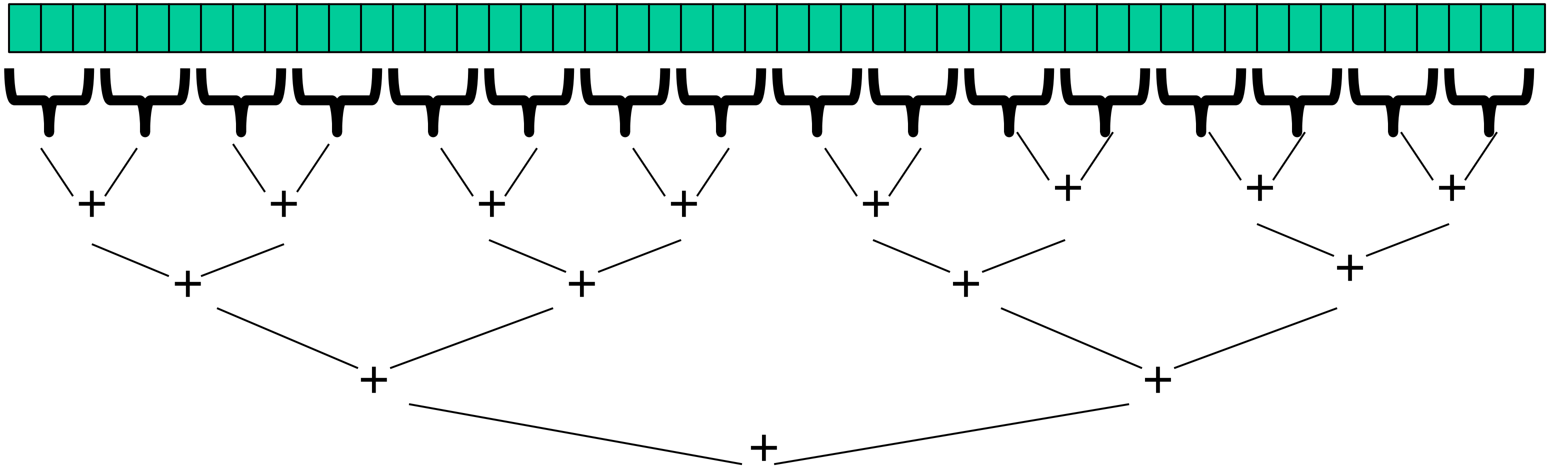
Time to divide up/recombine (linear loop version):

(n steps to perform, and each depends on the last)

Time to solve the subproblems (linear loop version):

(n steps to perform, independent of each other)

A better idea



The zombie apocalypse is straightforward using divide-and-conquer parallelism for the recursive calls

Note: a natural way to code it is to fork a bunch of tasks, join them, and get results.
But… the natural zombie way is to bite one human and then each “recurse”.
As is so often the case, the zombie way is better!

How *Do We Divide Up the Work?*

The metaphor is not perfect. Each time we “infect” a processor, it goes off and does useful work. However, the analysis still holds:

Let n be the array size and P be the number of processors.

Time to divide up/recombine (divide-and-conquer version):

(n steps to perform, arranged in a balanced tree)

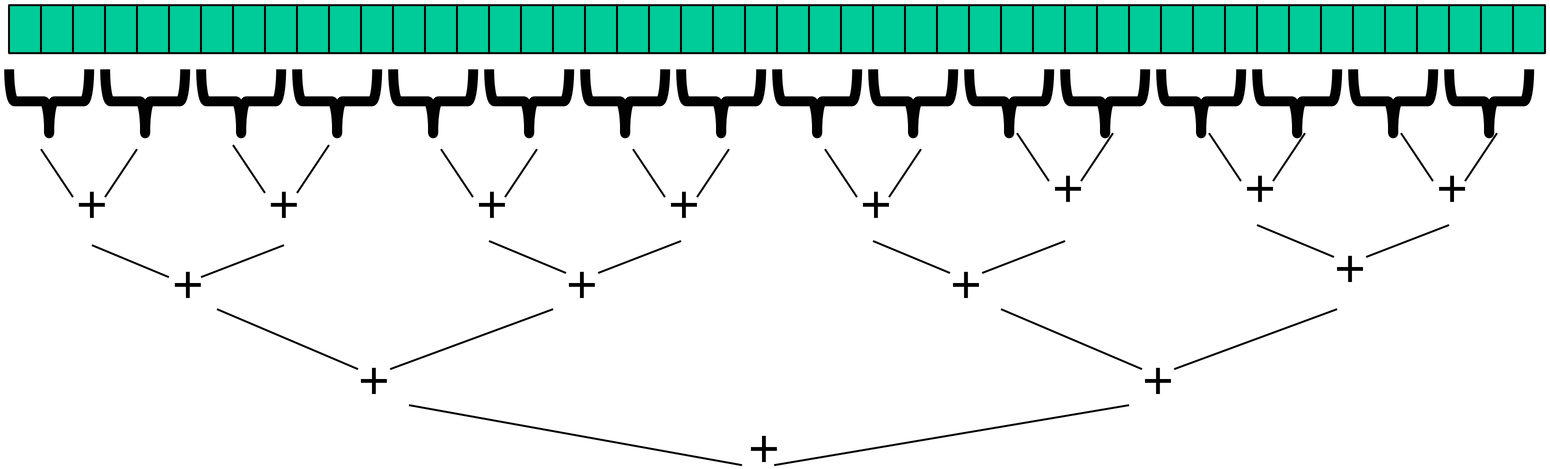
Time to solve the subproblems (divide-and-conquer version):

(n steps to perform, independent of each other)

Divide-and-conquer really works

- The key is divide-and-conquer parallelizes the result-combining
  - *If* you have enough processors, total time is height of the tree: $O(\texttt{log}\ n)$ (optimal, exponentially faster than sequential $O(n)$)
  - Next lecture: study reality of **P** << *n* processors

- Will write all our parallel algorithms in this style
  - But using a special library engineered for this style
    - Takes care of scheduling the computation well
  - Often relies on operations being associative (like +)



Being realistic

Creating one task per element *still* so expensive that it wipes out parallelism savings.

So, use a *sequential cutoff*, typically ~500-1000. (This is like switching from quicksort to insertion sort for small subproblems.)

Exercise: If there are 1,000,000 (~2^{20}) elements in the array and our cutoff is 1, about how many tasks do we create? (I.e., nodes in the tree.)

Exercise: If there are 1,000,000 (~2^{20}) elements in the array and our cutoff is 1,000 (~2^{10}), about how many tasks do we create?

That library, finally

- Even with all this care, C++11’s threads are usually too “heavyweight” (implementation dependent).
- OpenMP 3.0’s *main contribution* was to meet the needs of divide-and-conquer fork-join parallelism
  - Available in recent g++’s.
  - See provided code and notes for details.
  - Efficient implementation is a fascinating but advanced topic!

Example: final version

```
int cmp_helper(int array[], int len, int target) {
  const int SEQUENTIAL_CUTOFF = 1000;
  if (len <= SEQUENTIAL_CUTOFF)
    return count_matches(array, len, target);

  int left, right;
#pragma omp task untied shared(left)
  left = cmp_helper(array, len/2, target);
  right = cmp_helper(array+len/2, len-(len/2), target);
#pragma omp taskwait

  return left + right;
}

int cm_parallel(int array[], int len, int target) {
  int result;

#pragma omp parallel
#pragma omp single
  result = cmp_helper(array, len, target);

  return result;
}
```

OMP fork/join Cheat Sheet

- Just before a statement/block where you want parallelism:

  ```
  #pragma omp parallel
  #pragma omp single
  ```

- Just before a statement/block that is forking off a new task:

  ```
  #pragma omp task shared(…)
  ```

  where you list the result variables that are coming back.

- When you want to join (wait for) the other tasks:

  ```
  #pragma omp taskwait
  ```

- Pragmas are instructions to the compiler. Code will still run even if pragmas are ignored.

C++11 fork/join Cheat Sheet

- C++11 threads are much more expensive than OMP tasks, so you’ll need a **much** larger sequential cut-off.
- To fork a new thread, create a C++11 std::thread object and pass it the function to run in its own thread:

  ```
  std::thread foo;
  foo = std::thread(&function_name, arguments, …);
  ```

- When you want to join (wait for) a thread:

  ```
  foo.join();
  ```