CPSC 221
Basic Algorithms and Data Structures

A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency, Part 1

Introduction to Multithreading & Fork-Join Parallelism

Steve Wolfman, based on work by Dan Grossman
(with minor tweaks by Hassan Khosravi)

Learning Goals

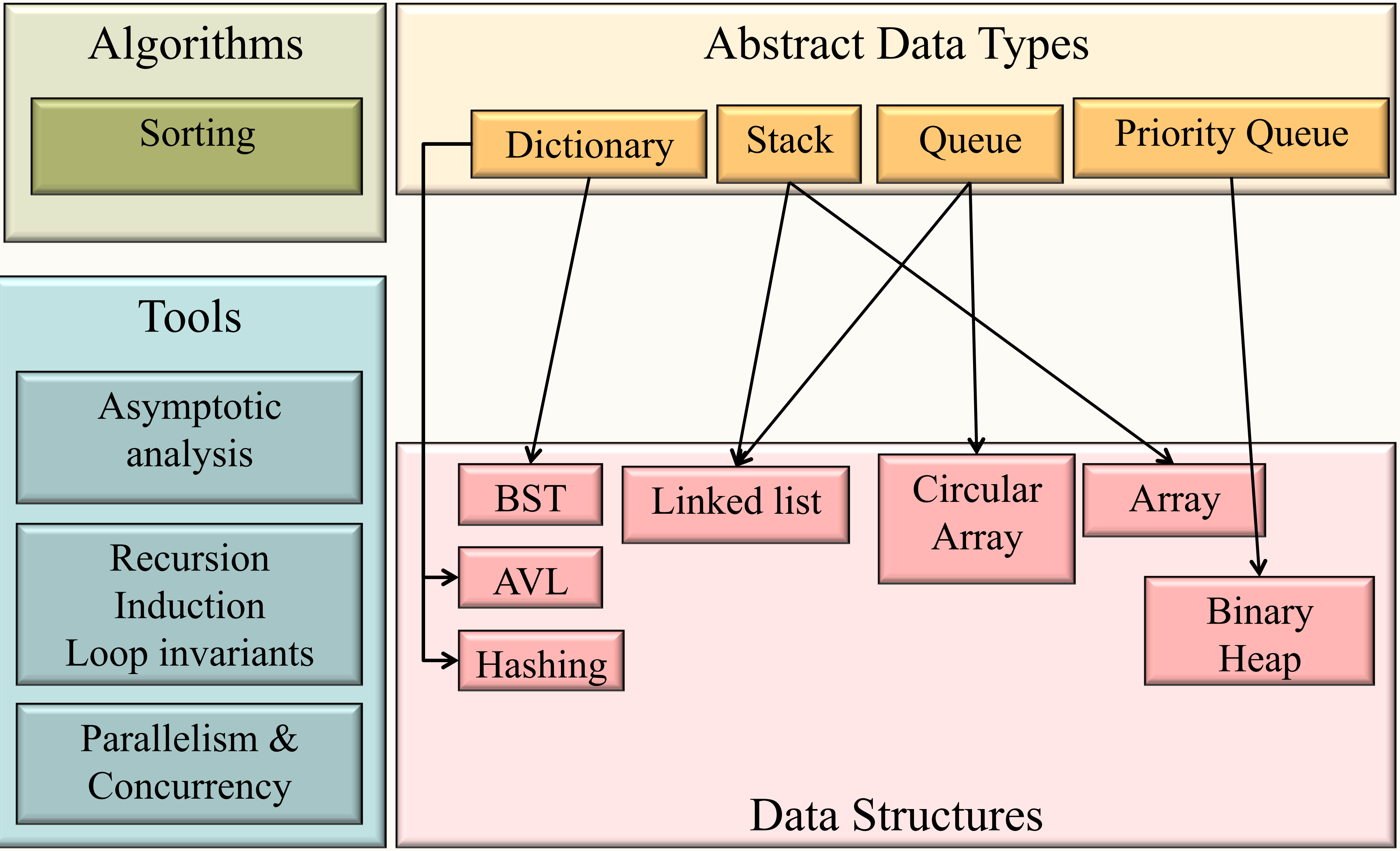
By the end of this unit, you should be able to:

- 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.)

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

CPSC 221 Journey

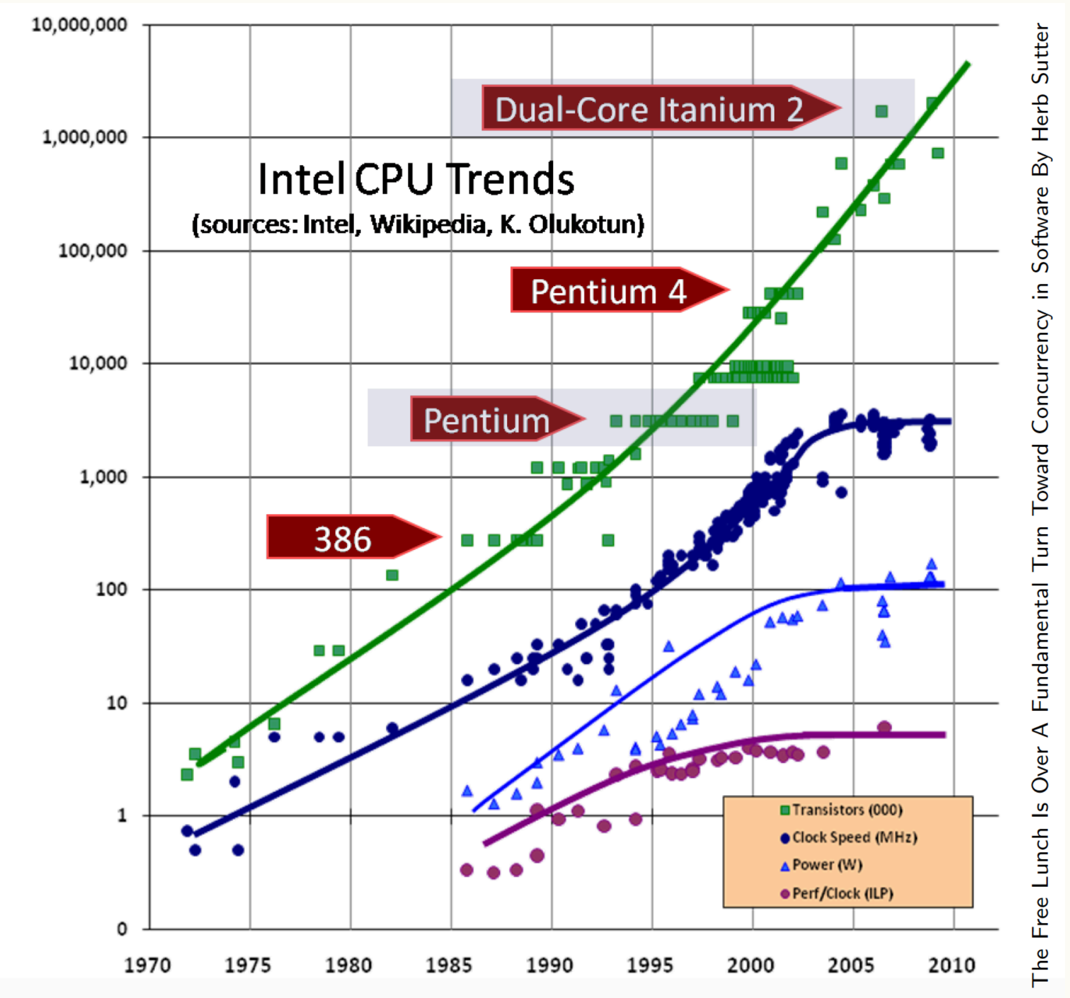


A simplified view of history

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.

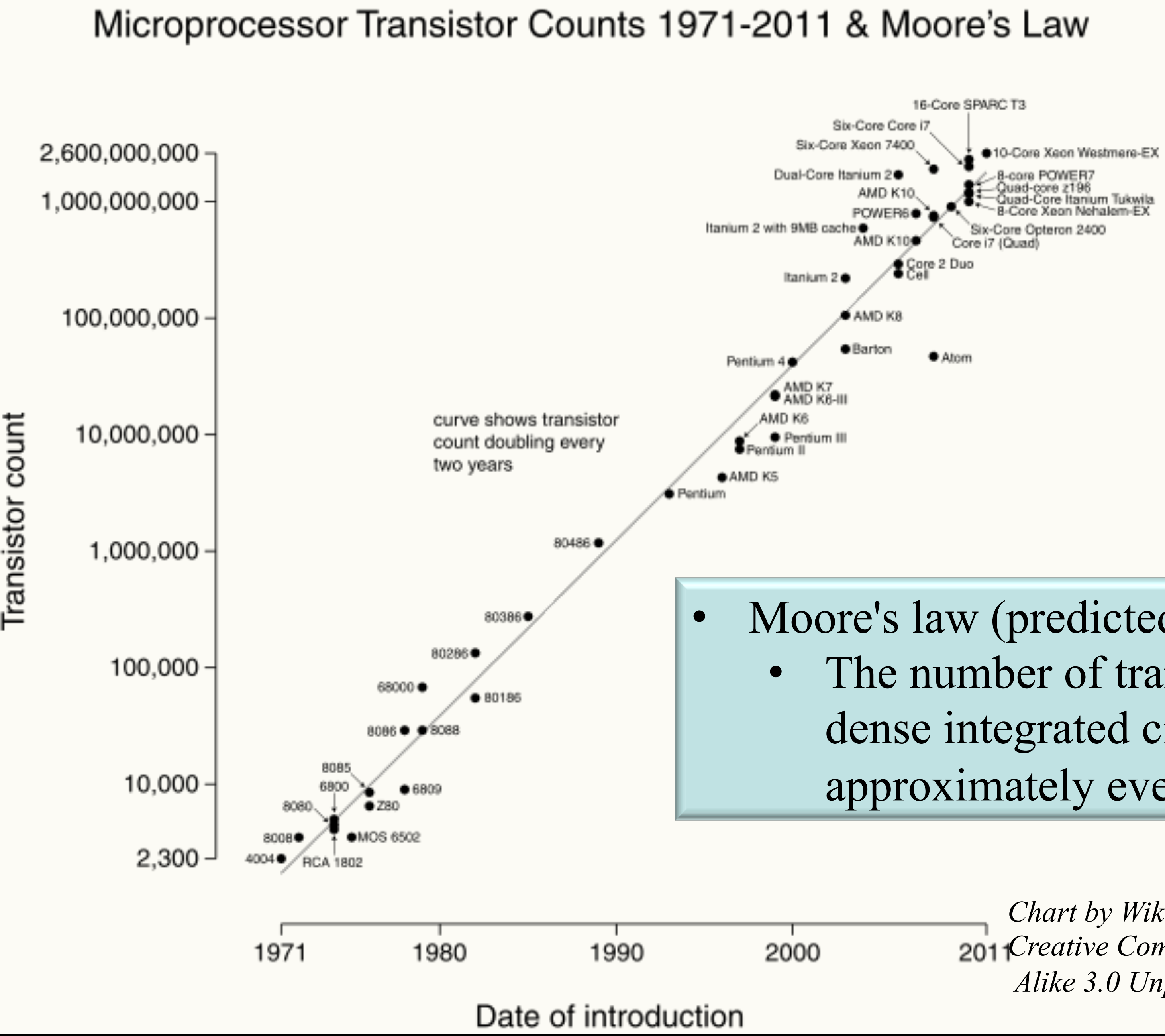
Properties of Intel CPUs



Although we keep making transistors 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

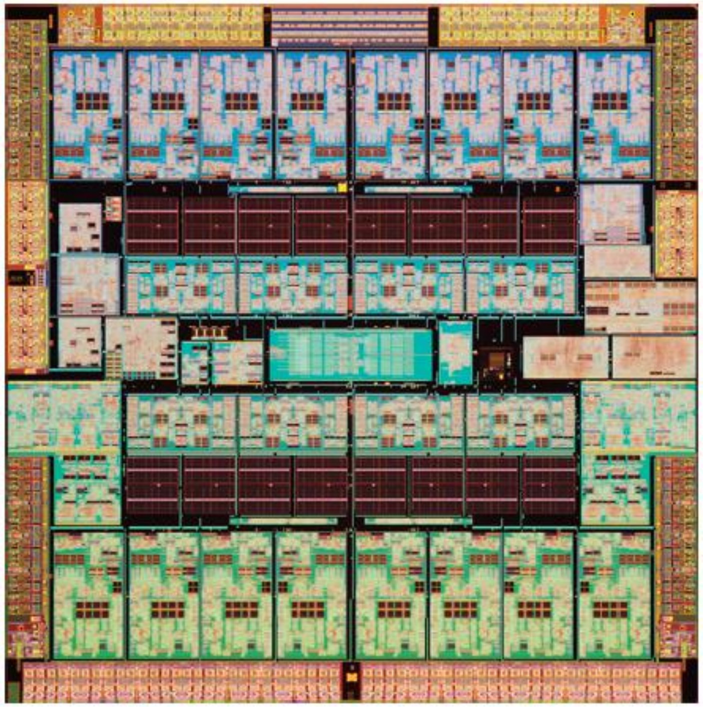
Solution, not faster but smaller and more…



- Moore's law (predicted in 1975)
  - The number of transistors in a dense integrated circuit doubles approximately every two years

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

Microprocessor Transistor Counts



Sparc T3 micrograph from Oracle; 16 cores.

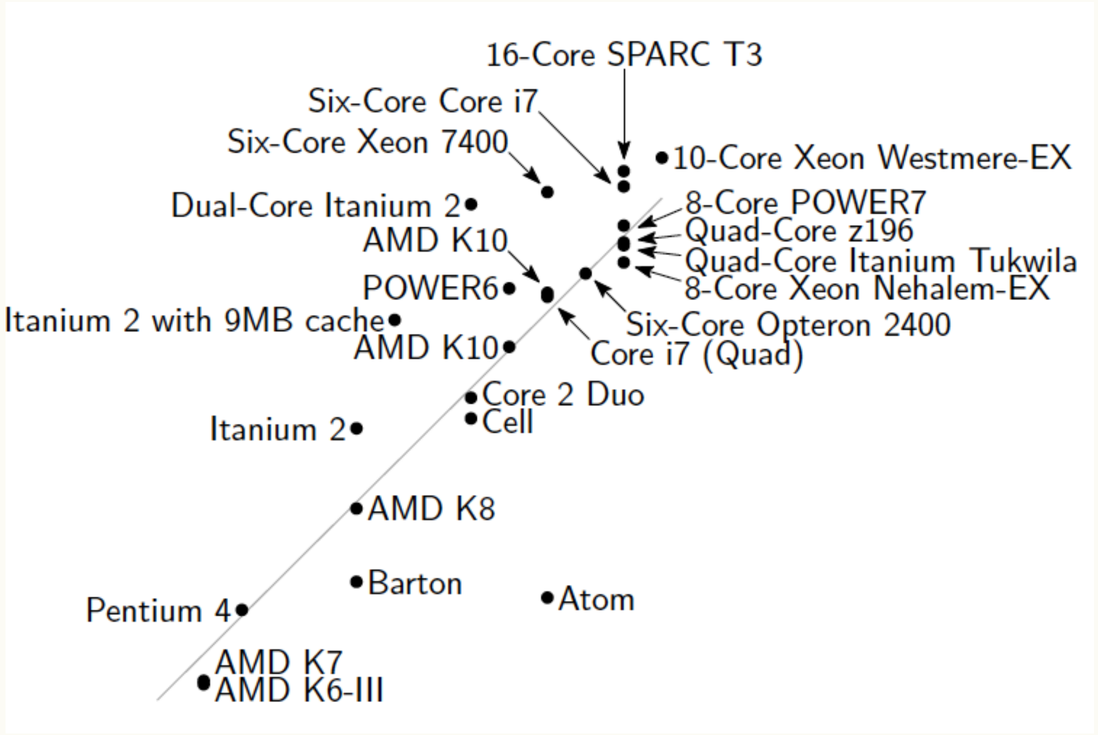


Chart by Wikimedia user: Wgsimon
Creative Commons Attribution-Share Alike 3.0 Unported

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

(Goodbye to) Sequential Programming

One thing happens at a time.

The next thing to happen is "my" next instruction.

Removing these assumptions creates challenges & opportunities:

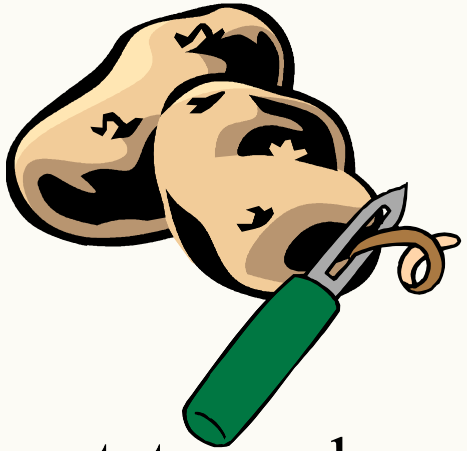
- How can we get more work done per unit time (throughput)?
- How do we divide work among threads of execution and coordinate (synchronize) among them?
- How do we support multiple threads operating on data simultaneously (concurrent access)?
- How do we do all this in a principled way? (Algorithms and data structures, of course!)

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?

~25 minutes + Setup cost + distribution of recourses

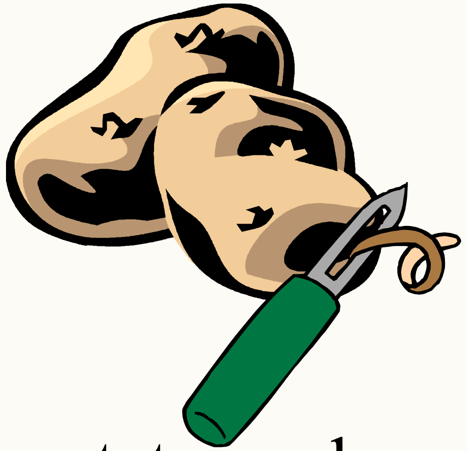
Parallelism: using extra resources to solve a problem faster.

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?

~25 minutes + Setup cost + distribution of recourses

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

Problem: Count Matches of a Target

- How many times does the number 3 appear?

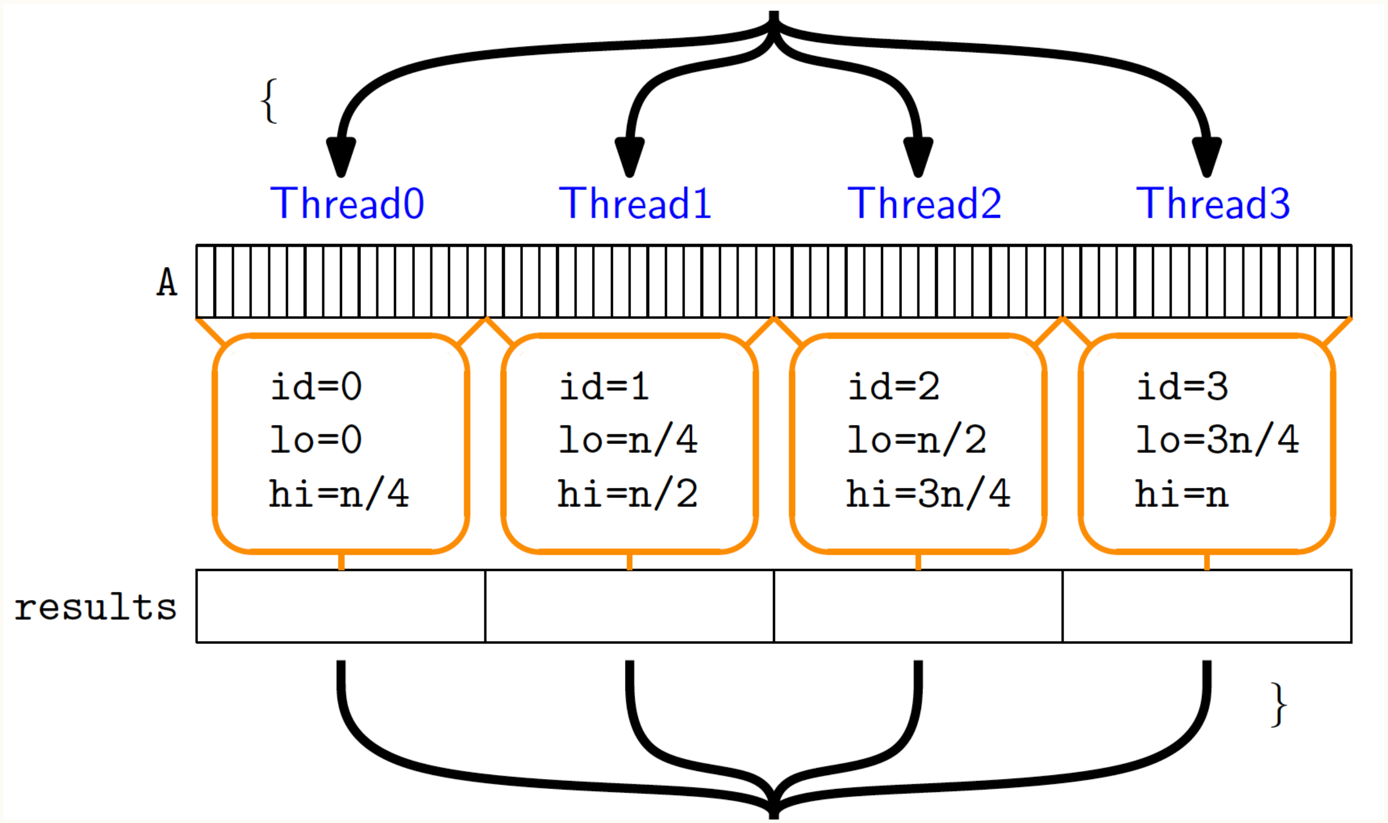
3	5	9	3	2	0	4	6	1	3

```c
// 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?

Parallelism Example

- Parallelism: Use extra computational resources to solve a problem faster (increasing throughput via simultaneous execution)
- General idea



Parallelism Example

- Parallelism: Use extra computational resources to solve a problem faster (increasing throughput via simultaneous execution)
- Pseudocode for counting matches
  - Bad style for reasons we'll see, but may get roughly 4x speedup

```
int cm_parallel(int arr[], int len, int target){
  res = new int[4];
  FORALL(i=0; i < 4; i++) { //parallel iterations
    res[i] = count_matches(arr + i*len/4,
                           (i+1)*len/4 – i*len/4, target);
 }
  return res[0]+res[1]+res[2]+res[3];
}

int count_matches(int arr[], int len, int target) {
  // normal sequential code to count matches of target.
}
```

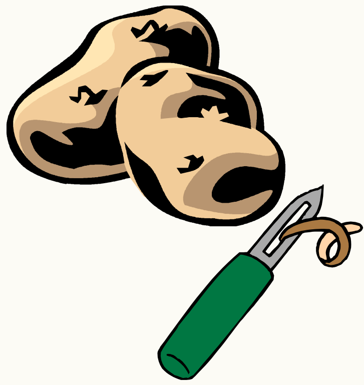
KP Duty: Peeling Potatoes, Concurrency

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 4 people with 3 potato peelers to peel 10,000 potatoes?

(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.)



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 4 people with 3 potato peelers to peel 10,000 potatoes?

Concurrency: Correctly and efficiently manage access to shared resources

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

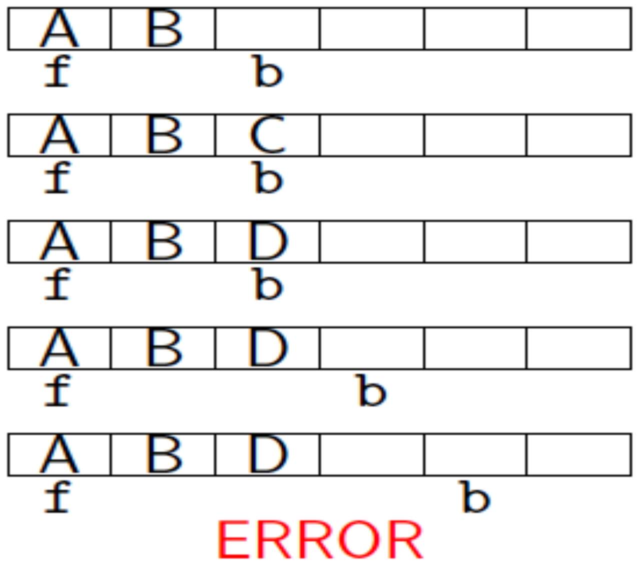
Concurrency Example

Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)

Pseudocode for a shared Queue

```
void enQ(Obj x) {
  Q[b] = x;
  b=(b+1)%n;
}
```

time	Thread1	Thread2
0	enQ("C")	enQ("D")
1	Q[b]="C"	
2		Q[b]="D"
3	b=(b+1)%n	
4		b=(b+1)%n



Concurrency will be covered in CPSC 213

Concurrency Example

Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)

Pseudocode for a shared chaining hashtable

- Prevent *bad interleavings* (correctness)
- But allow some concurrent access (performance)

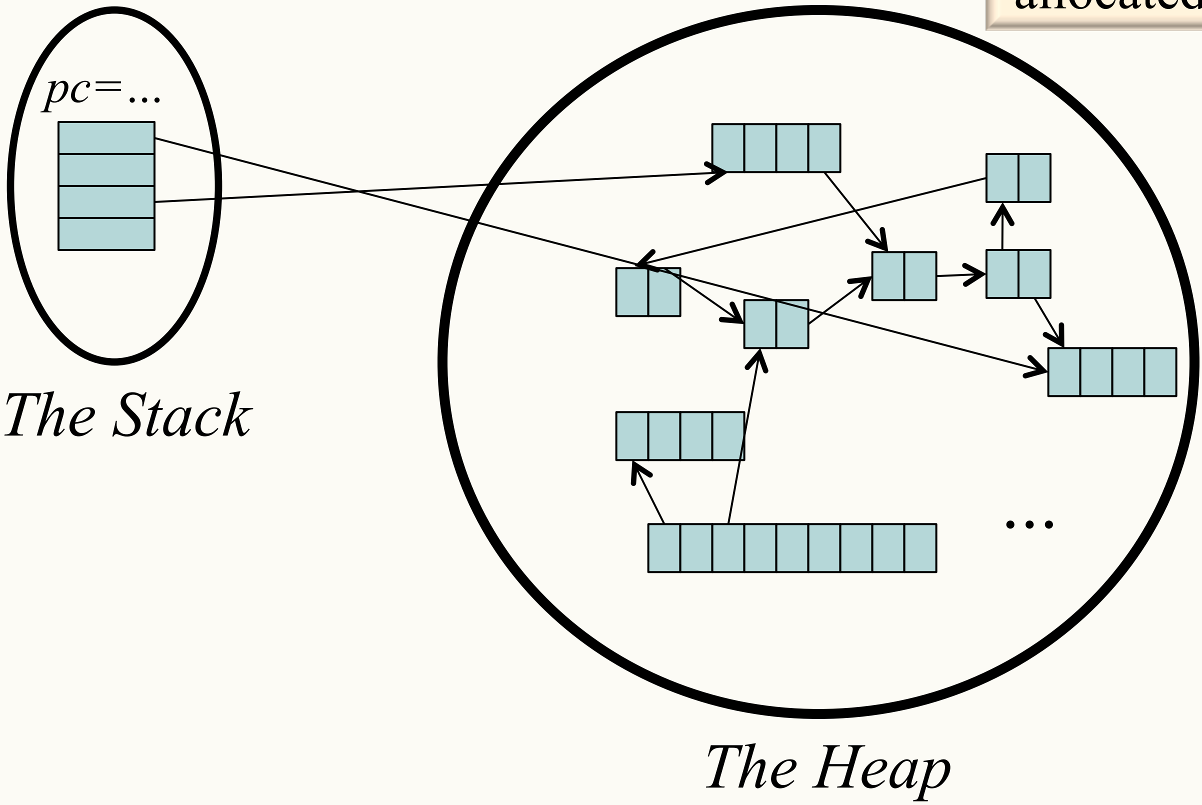
```
template <typename K, typename V>class Hashtable<K,V> {
void insert(K key, V value) {
    int bucket = …;
    prevent-other-inserts/lookups in table[bucket]
    do the insertion
      re-enable access to table[bucket]
      }
  V lookup(K key) {
    (like insert, but can allow concurrent
     lookups to same bucket)
  }
}
```

Concurrency will be covered in CPSC 213

OLD Memory Model

Dynamically allocated data.

Local variables
Control flow info



(pc = program counter, address of current instruction)

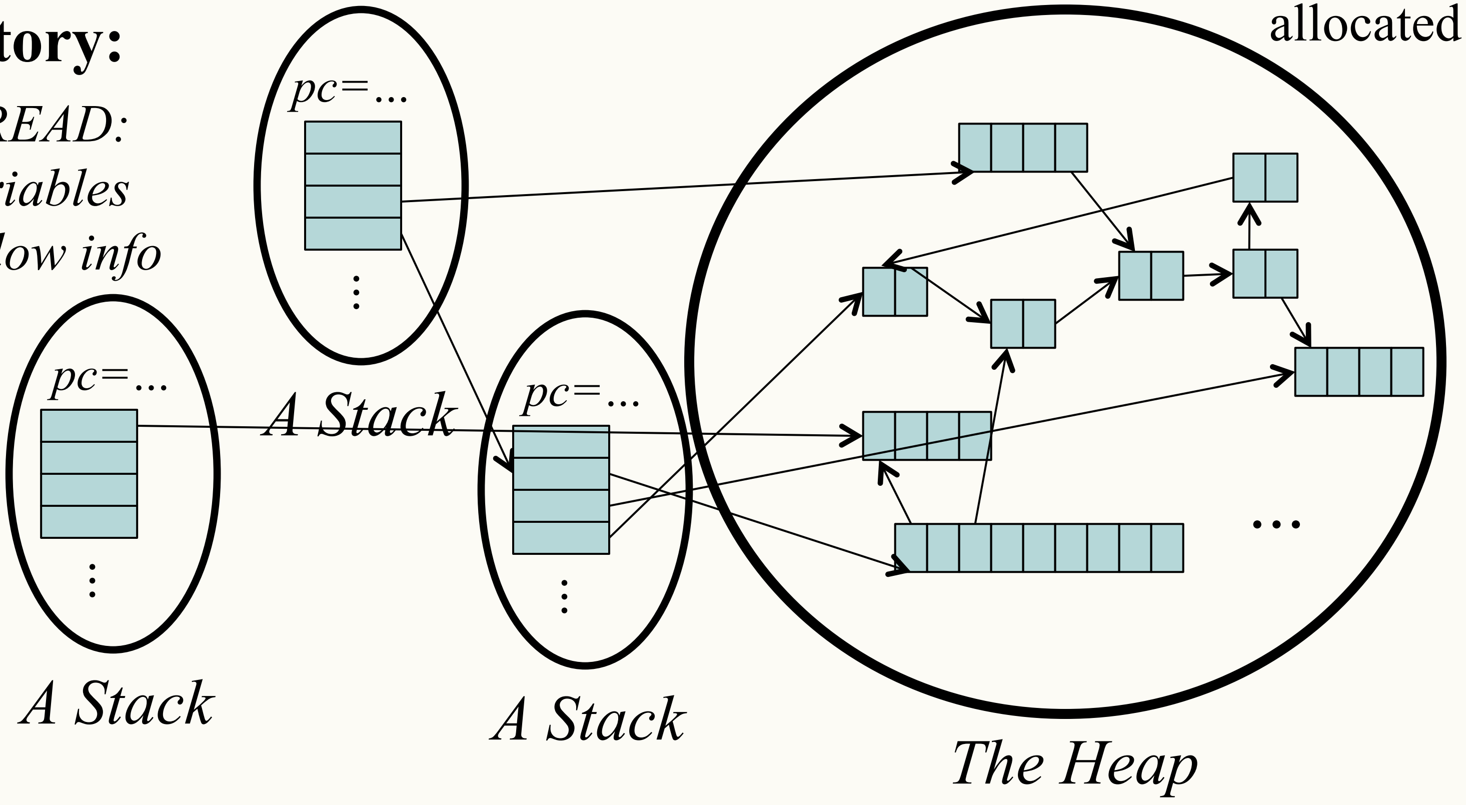
Shared Memory Model

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

NEW story:

PER THREAD:
Local variables
Control flow info

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"

Note: our parallelism solution will have a "dataflow feel" to it.

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 = 4;

  std::thread workers[divs];
  int results[divs];
  for (int d = 0; d < divs; d++)
        workers[d] = std::thread(&cmp_helper, &results[d], array,
        (d*len)/divisions, ((d+1)*len)/divisions, 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! BE CAREFUL sharing memory!

```
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 = 4;

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

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

  return matches;
}
```

Race condition: What happens if one thread tries to write to a memory location while another reads (or multiple try to write)? KABOOM (possibly silently!)

```
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 = 4;

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

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

  return matches;
}
```

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??)

```
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 = 4;

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

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

  return matches;
}
```

Now, let's run it.

KABOOM! What happens, and how do we fix it?

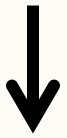
Join (not the most descriptive word)

- 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"

Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own

– **The constructor calls its argument *in a new thread* (forks)**



Fork/Join Parallelism

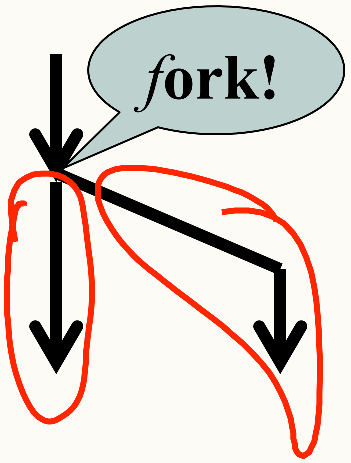
`std::thread` defines methods you could not implement on your own

- **The constructor calls its argument *in a new thread* (forks)**

Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own

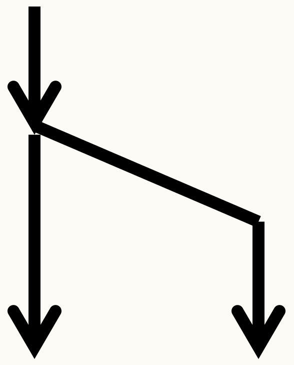
– **The constructor calls its argument *in a new thread* (forks)**



Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own

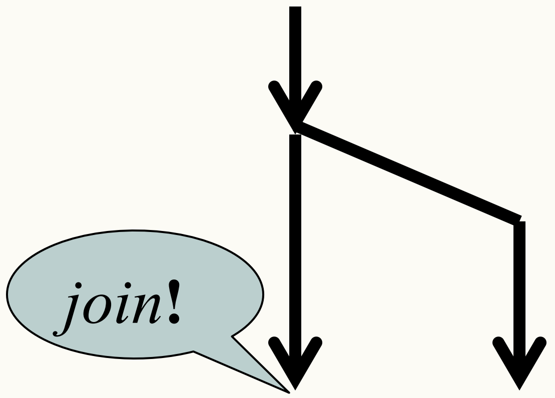
- **The constructor calls its argument *in a new thread* (forks)**



Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own

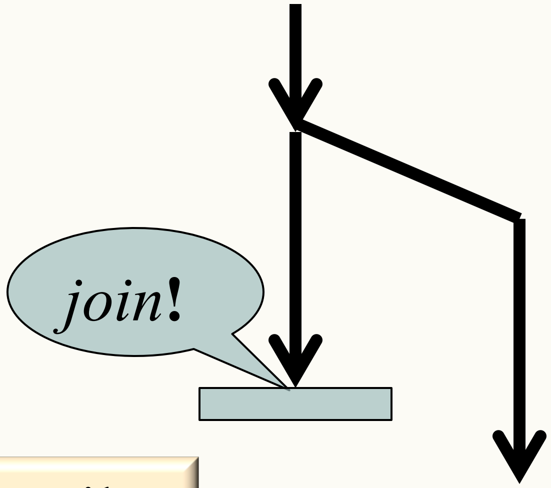
- The constructor calls its argument *in a new thread* (forks)
- **`join` blocks until/unless the receiver is done executing (i.e., its constructor's argument function returns)**



Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own

- The constructor calls its argument *in a new thread* (forks)
- **`join` blocks until/unless the receiver is done executing (i.e., its constructor's argument function returns)**

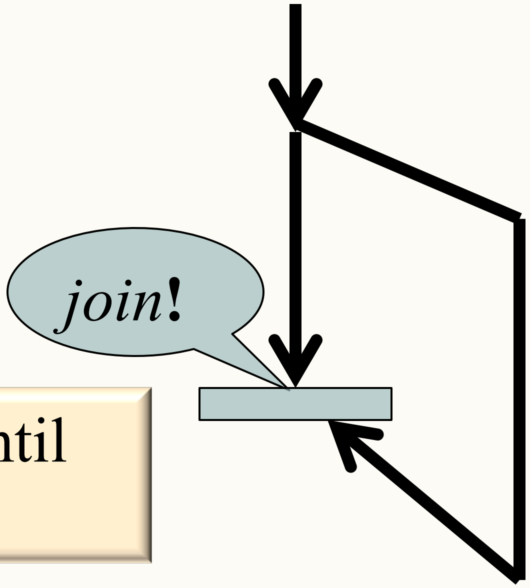


This thread is stuck until the other one finishes.

Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own

- The constructor calls its argument *in a new thread* (forks)
- **`join` blocks until/unless the receiver is done executing (i.e., its constructor's argument function returns)**



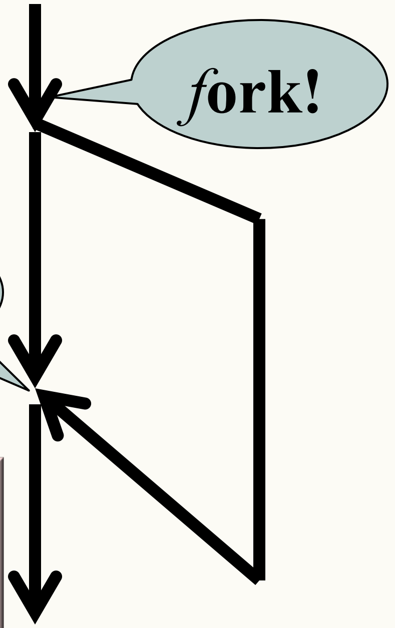
This thread could already be done (joins immediately) or could run for a long time.

This thread is stuck until the other one finishes.

Join

`std::thread` defines methods you could not implement on your own

- The constructor calls its argument *in a new thread* (forks)
- **`join` blocks until/unless the receiver is done executing (i.e., its constructor's argument function returns)**



And now the thread proceeds normally.

CPSC 221 Administrative Notes

- Lab 10 Parallelism Mar 26 – Apr 2
  - Some changes to the code since Friday
  - Marking Apr 7 – Apr 10 (Also doing Concept Inventory). Doing the Concept inventory is worth 1 lab point (0.33% course grade).
- Programming project #2 due
  - Apr Tue, 07 Apr @ 21.00
  - There was a typo in make, posted on Piazza
- PeerWise Call #5 due Apr 2 (5pm)
  - The deadline for contributing to your "Answer Score" and "Reputation score" is Monday April 20.

So… Where Were We?

- We talked about
  - Parallelism
  - Concurrency (mostly 213's problem)
- Problem: Count Matches of a Target
  - Race conditions
  - Out of scope variables
- Fork/Join Parallelism

Second attempt (patched!)

```
int cm_parallel(int array[], int len, int target) {
  int divs = 4;

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

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

  return matches;
}
```

Let’s run the code again!

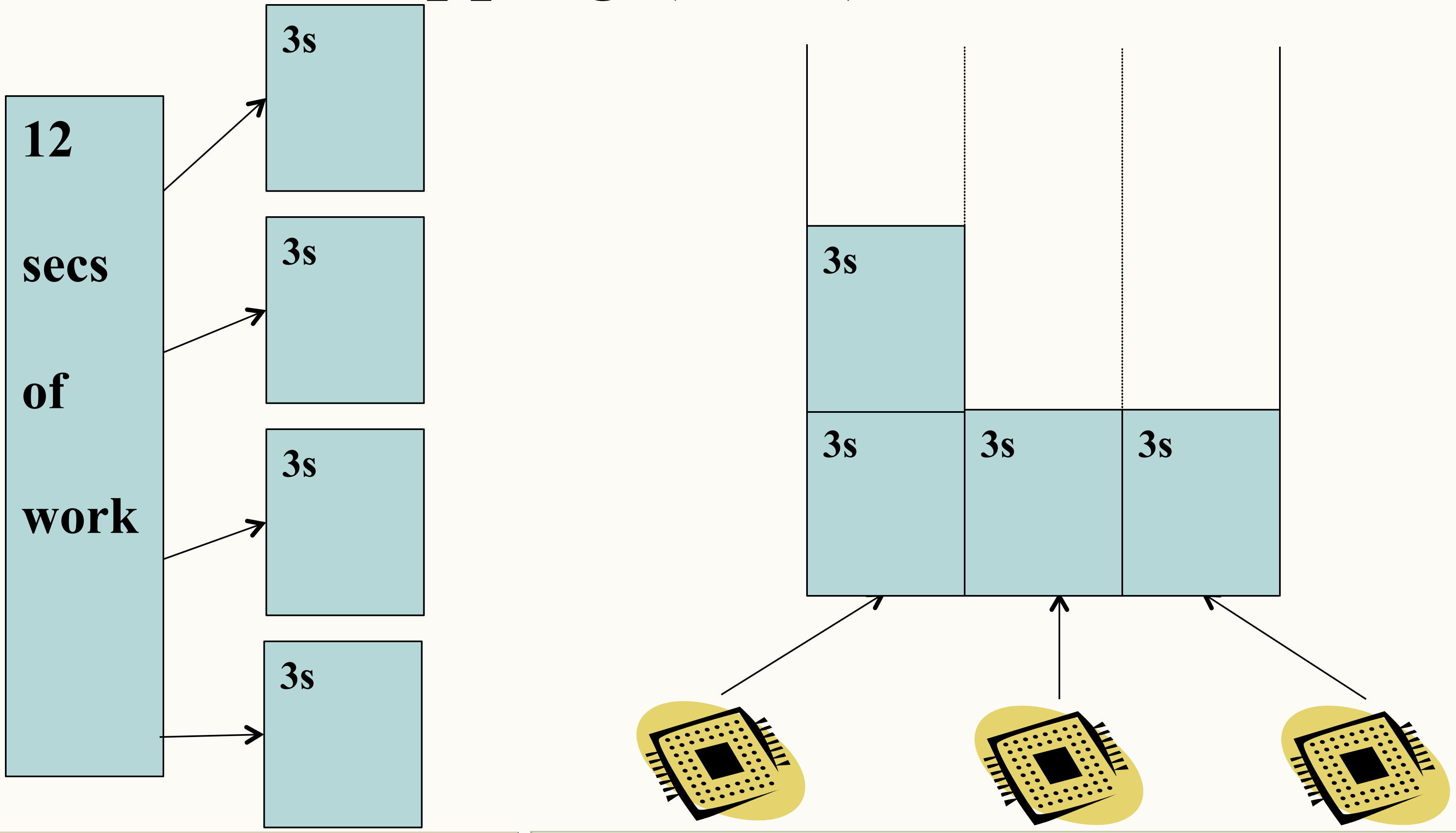
Success! Are we done?

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?

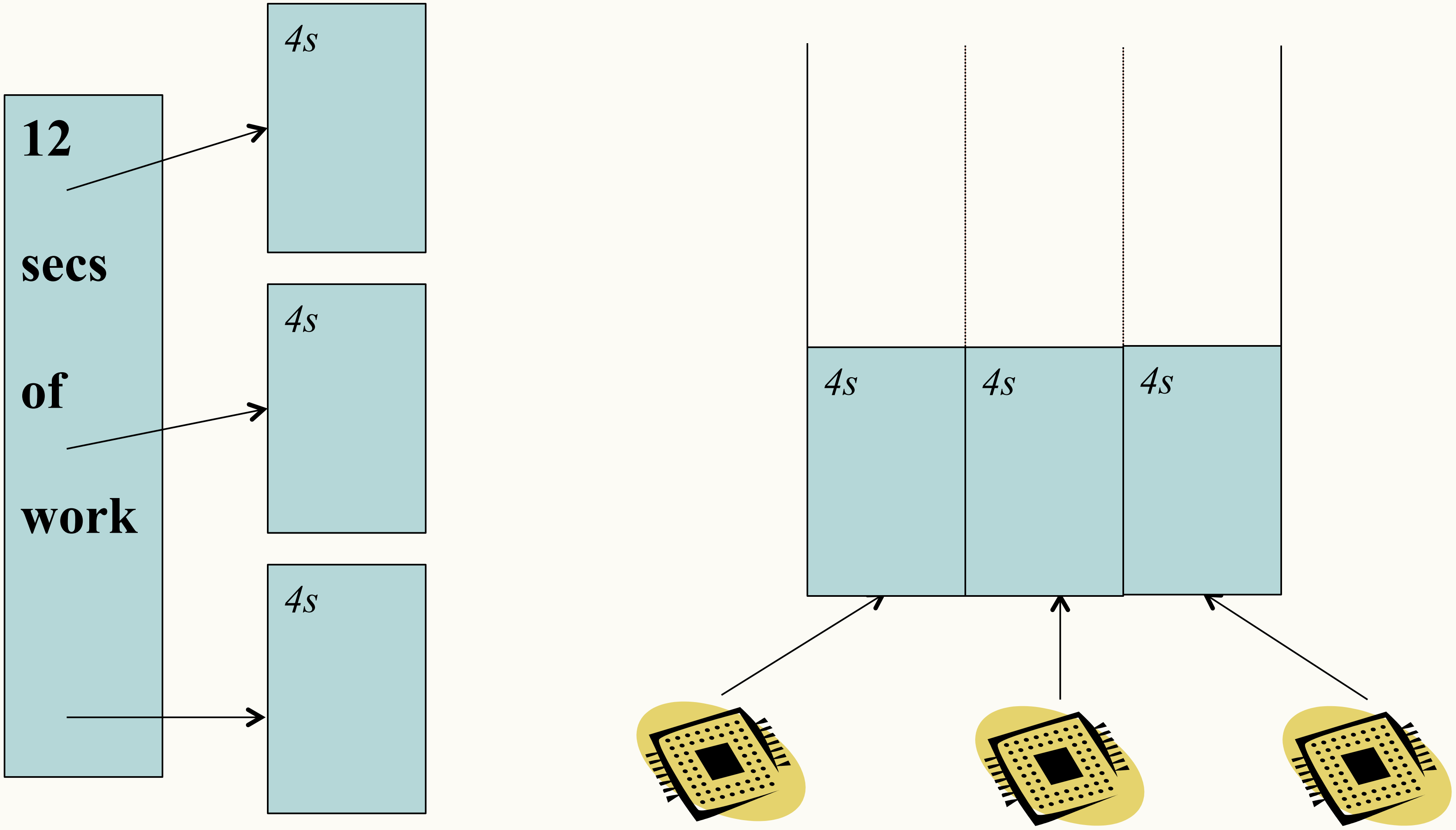
Chopping (a Bit) Too Fine



We thought there were 4 processors available.

But there's only 3. Result?
Would take ~ twice as long as with 4 processors

Chopping (a Bit) Too Fine



We thought there were 3 processors available.

And there are. Results?

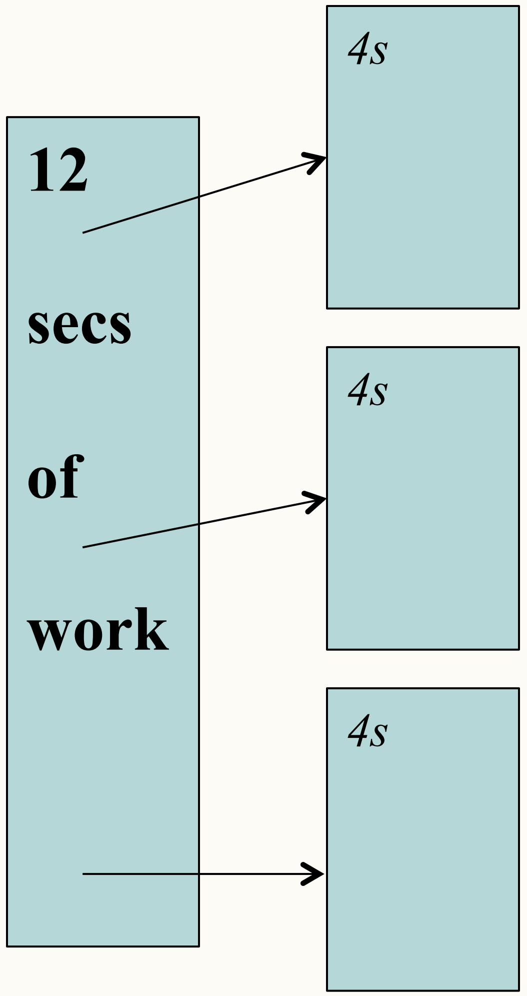
Chopping (a Bit) Too Fine

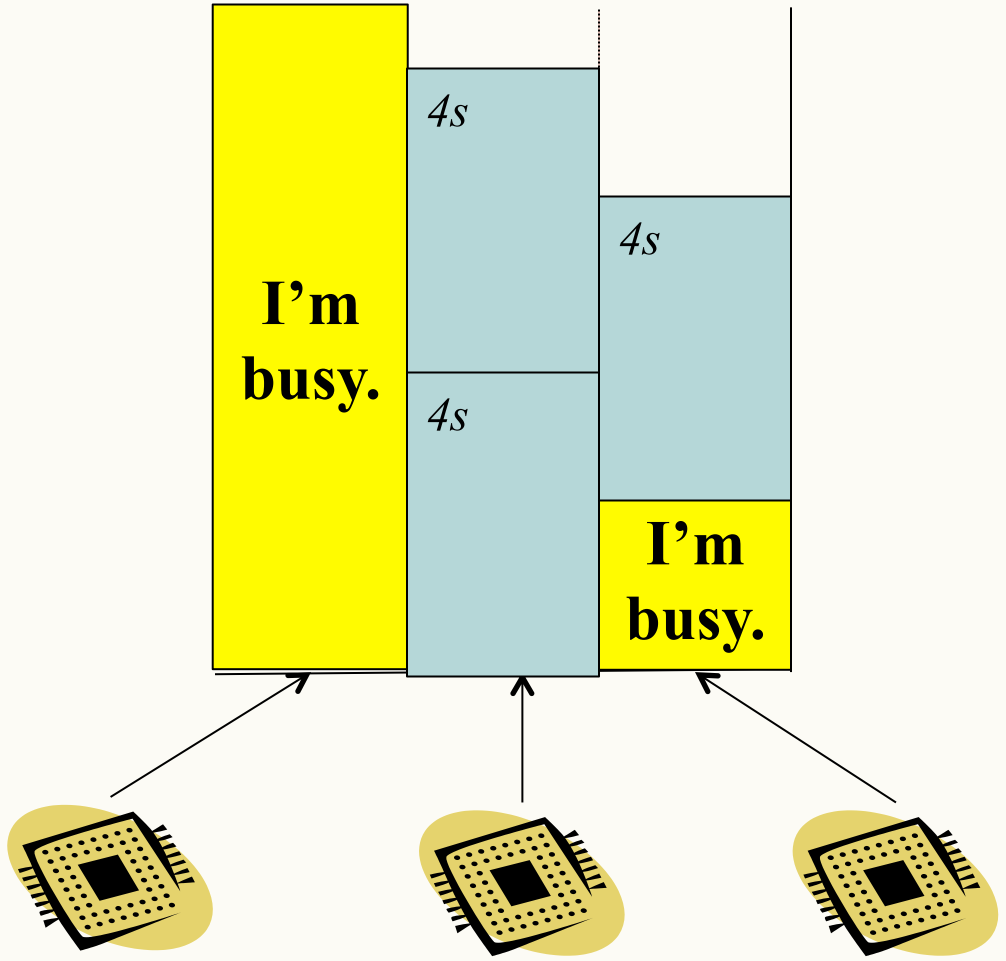
- Do not use constants where a variable is appropriate.

(Note: I can use the following to get # cores
- std::thread::hardware_concurrency()
- omp_get_num_procs().

- So if I have p processors, should I just p threads?

Is there a "Just Right"?





We thought there were 3 processors available.

And there are, sort of. Results?

Chopping (a Bit) Too Fine

- Do not use constants where a variable is appropriate.

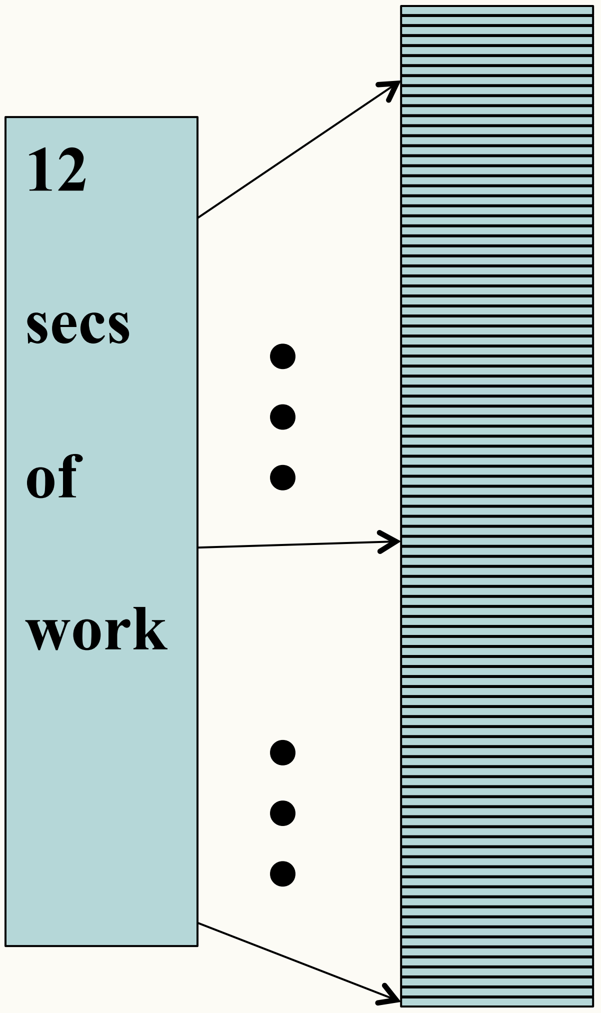
(Note: I can use the following to get # cores
- std::thread::hardware_concurrency()
- omp_get_num_procs().

- So if I have p processors, should I just p threads?

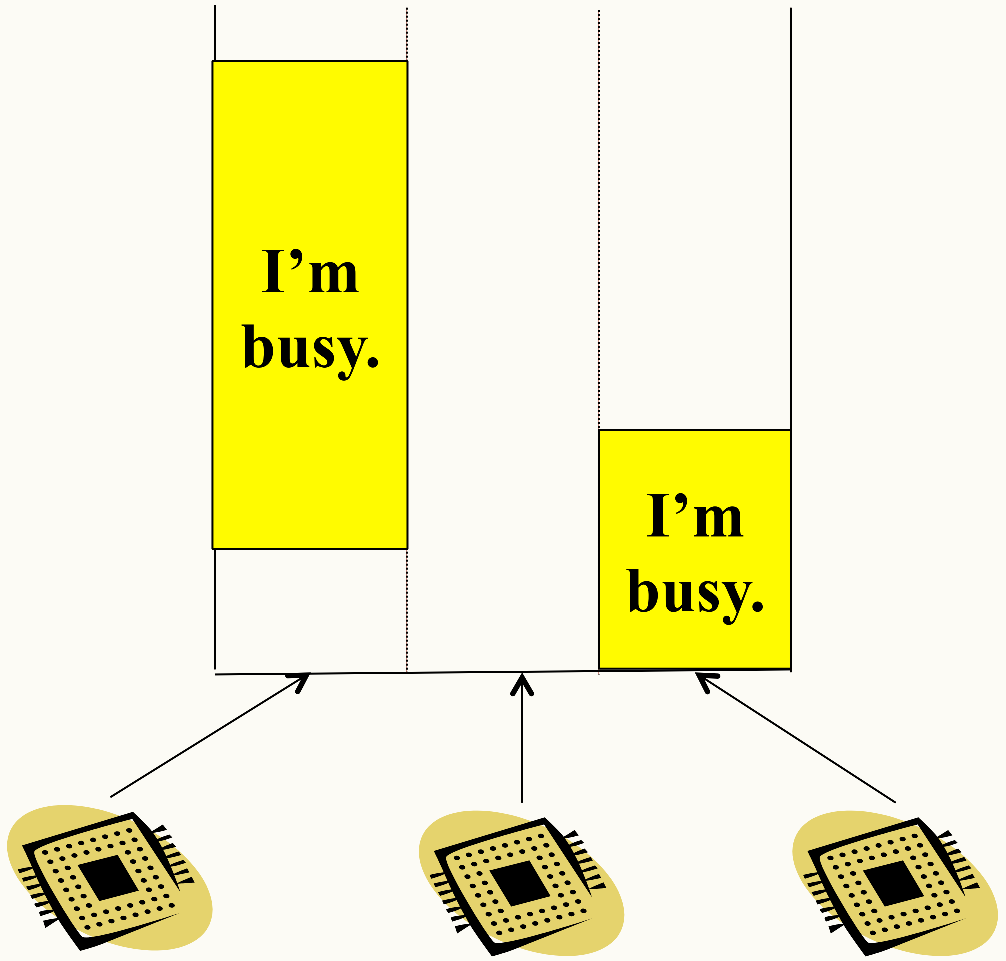
We cannot always assume that every processor is available to the code we're writing all the time.

We cannot predictably divide the work into approximately equal pieces (processing equal-size chunks of the array doesn't necessarily take the same amount of time.

Chopping So Fine It's Like Sand or Water



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



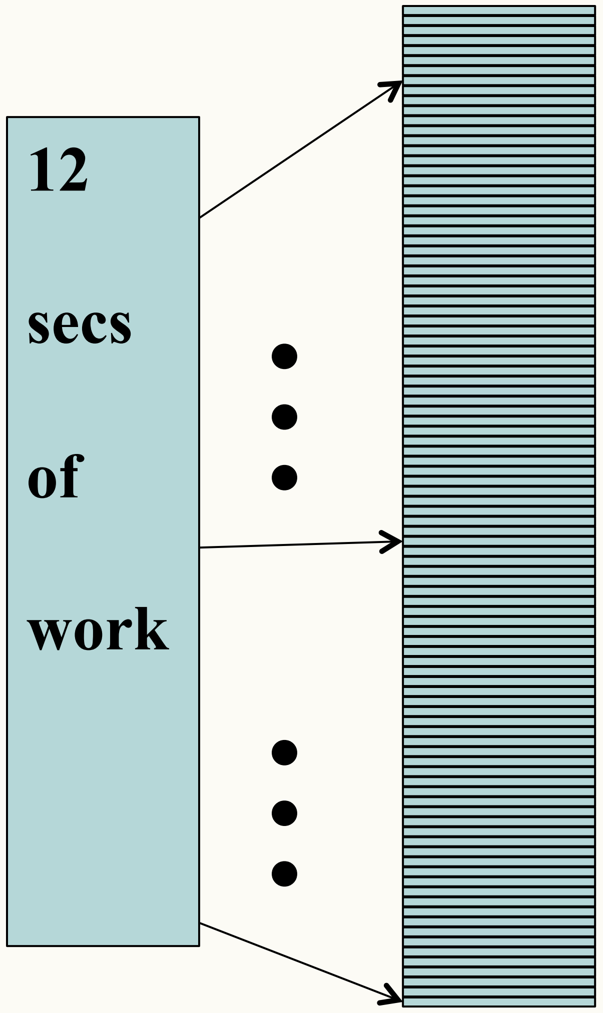
We chopped into 10,000 pieces.

And there are, sort of. Results?

Chopping So Fine It's Like Sand or Water

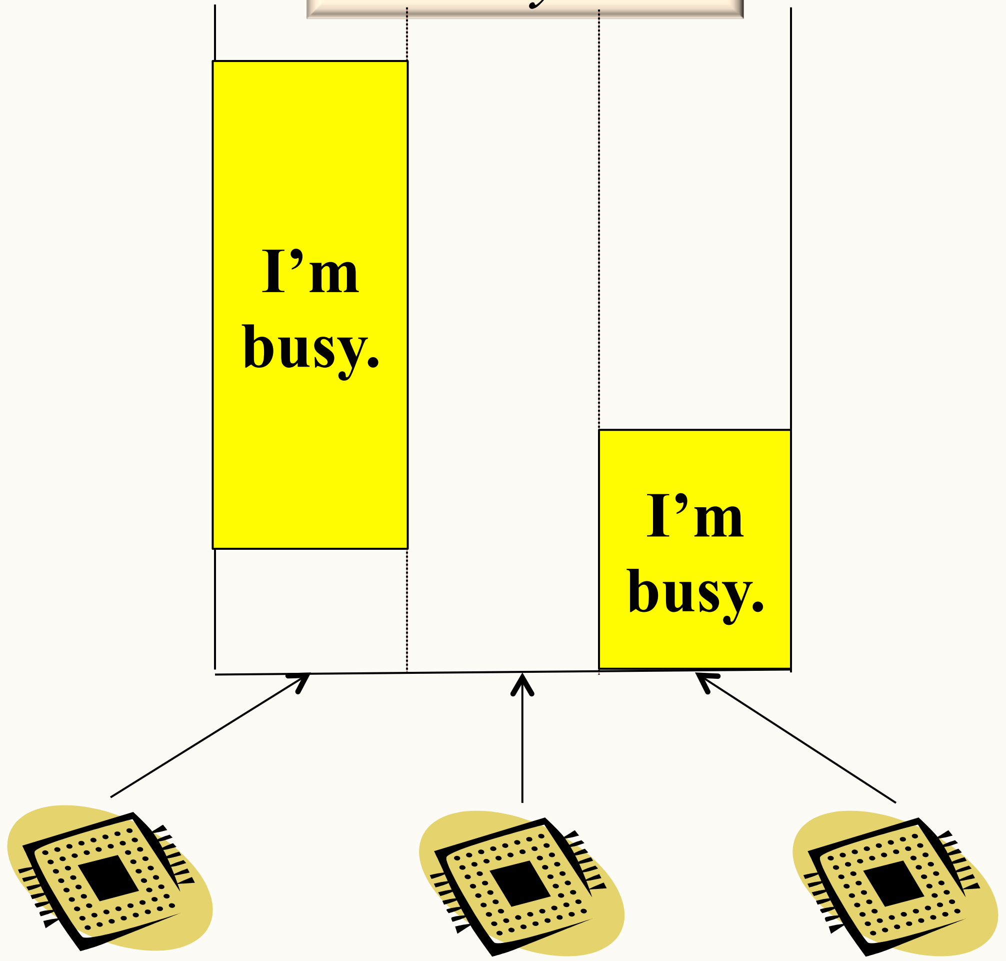
- This solves a few problems
  - any number of processors will stay busy until the very end
  - we just have a "big pile" of threads waiting to run. If the number of processors available changes, that affects only how fast the pile is processed, but we are always doing useful work with the resources available.
  - smaller chunks of work make load imbalance far less likely since the threads do not run as long. Also, if one processor has a slow chunk, other processors can continue processing faster chunks.

Chopping So Fine It's Like Sand or Water



We chopped into 10,000 pieces.

Let's try this



And there are, sort of. Results?

Analyzing Performance

```
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 = len;

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

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

  return matches;
}
```

$\Theta(n)$

$\Theta(1)$

$\Theta(1)$

It's Asymptotic Analysis Time!
(len=4 , # of processors = ∞)

Runtime $\in \Theta(n)$

Analyzing Performance

```
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 = len;

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

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

  return matches;
}
```

$\Theta(1)$

$\Theta(n)$

$\Theta(n)$

It's Asymptotic Analysis Time!
(len=n, # of processors = ∞)

Runtime $\in \Theta(n)$
That sucks!

Analyzing Performance

```
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 = len;

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

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

  return matches;
}
```

$\Theta(n/p)$

$\Theta(p)$

$\Theta(p)$

It's Asymptotic Analysis Time!
(len=P , # of processors = ∞)

Runtime $\in \Theta(n/P + P)$
What should P be?

Chopping So Fine It's Like Sand or Water

- This approach still has two major problems
  - C++11 threads: The threads will work and produce the correct answer, but the constant-factor overheads of creating each thread is far too large.
    - We need an implementation of threads that is designed for this kind of fork/join programming.
  - We now have more results to combine, which is time consuming.
    - we need a better way to combine results.

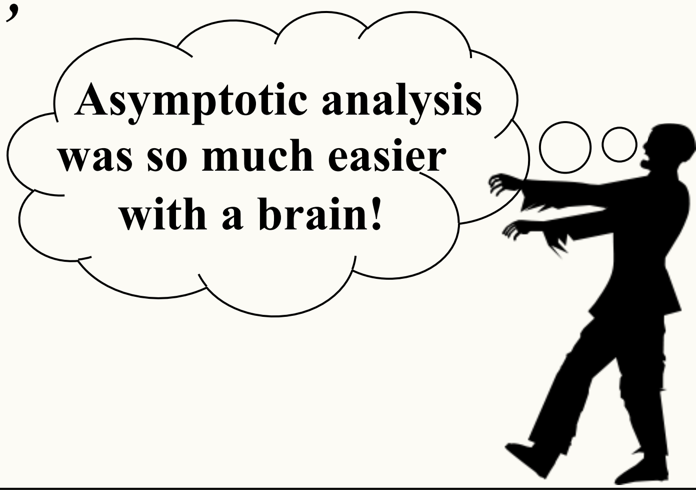
Zombies Seeking Help

A group of (non-CSist) zombies wants your help infecting the living. Each time a zombie bites a human, it gets to transfer a program.

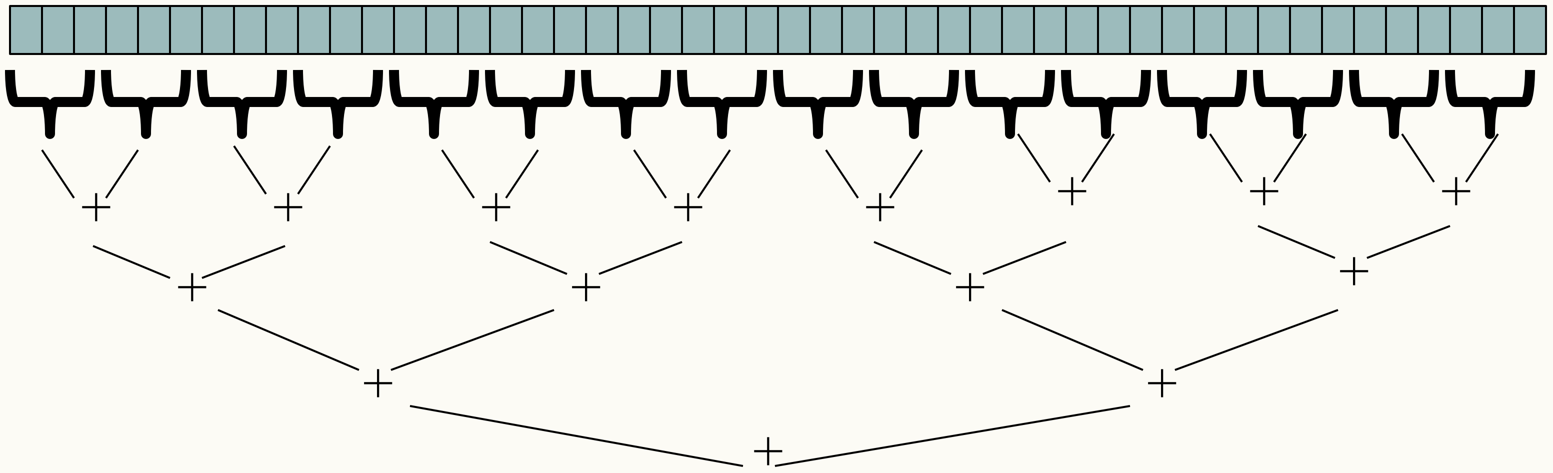
The new zombie in town has the humans line up and bites each in line, transferring the program: *Do nothing except say "Eat Brains!!"*

Analysis?

How do they do better?



A better idea



The zombie apocalypse is straightforward using divide-and-conquer

Divide-and-Conquer Style Code

still with a few problems

```
void cmp_helper(int * result, int array[], int lo, int hi, int target)
{
  if (len <= 1) {
    *result = count_matches(array + lo, hi-lo, target);
    return;
  }

  int left, right;
  int mid = lo + (hi-lo)/2;
  std::thread left(&cmp_helper, &left, array, lo, mid, target);
  std::thread right(&cmp_helper, &right, array, mid, hi, target);
  left.join();
  right .join();
  return left + right;
}

int cm_parallel(int array[], int len, int target) {
  int result;
  cmp_helper(&result, array, 0, len, target);
  return result;
}
```

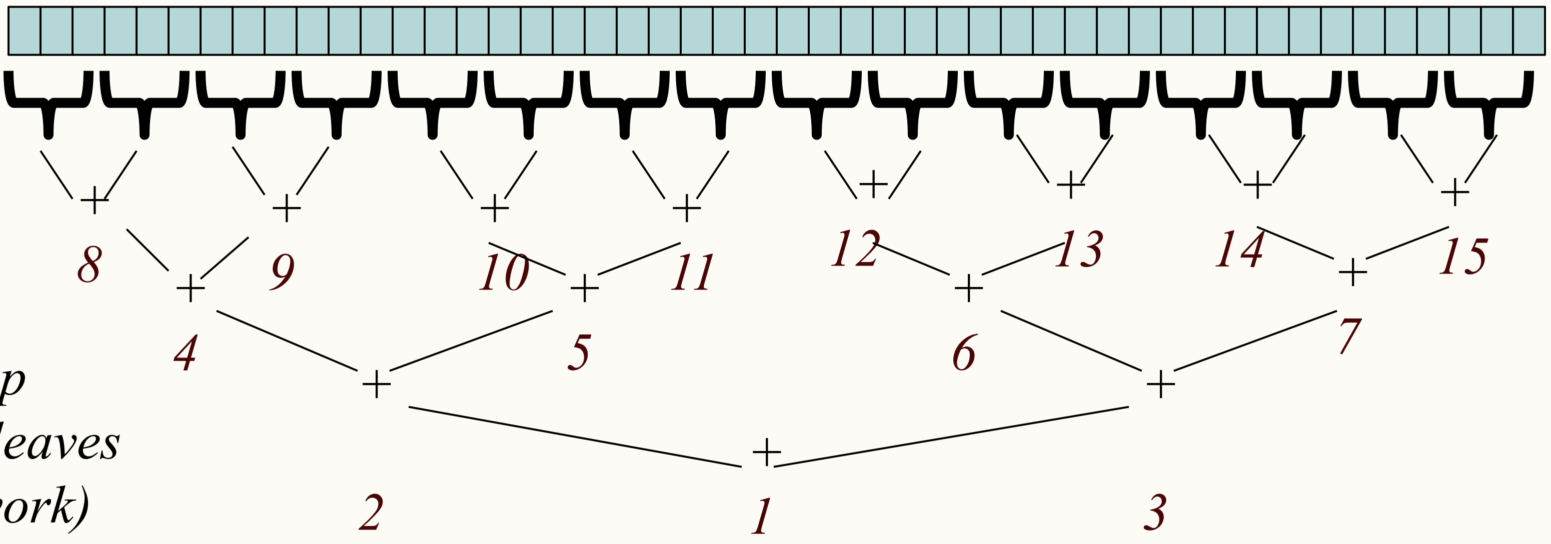
Lazy lazy threads!

- Each thread creates two helper threads left and right and then waits for them to finish and add the results.
- Rather than having all these threads wait around, it is more efficient to create *one helper thread* to do half the work and have the thread do the other half *itself*.

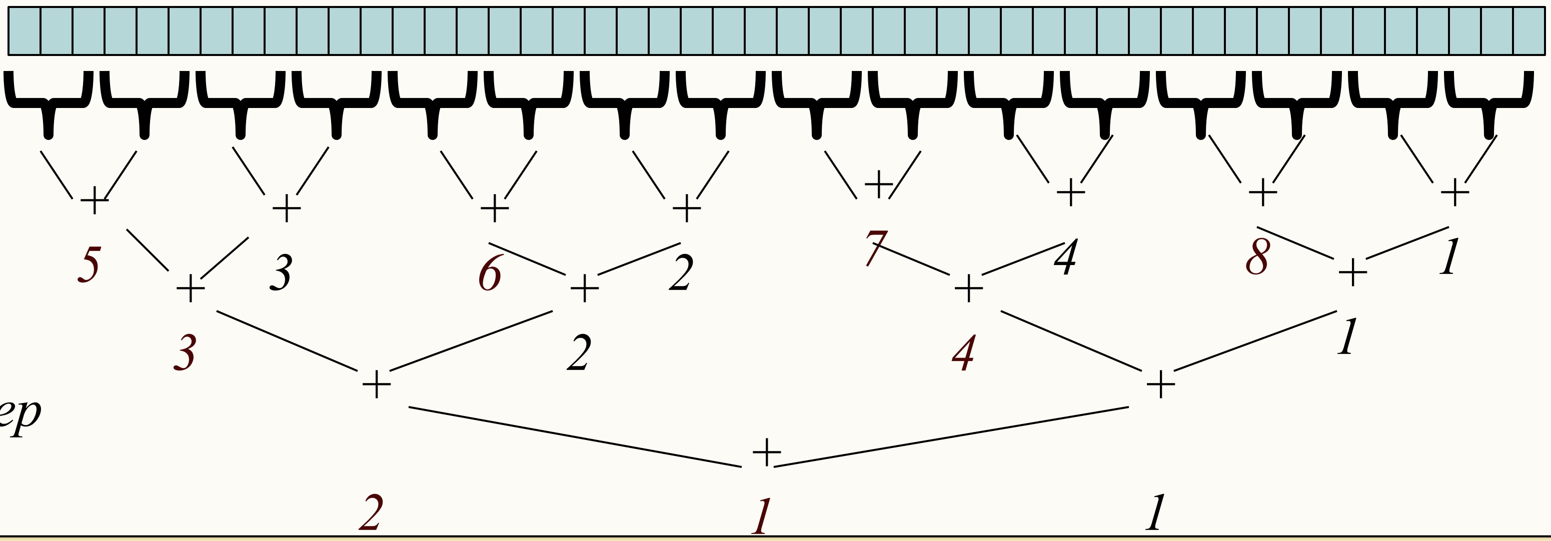
Reduces the number of threads by a factor of 2

Fewer threads pictorially

2 new threads at each step (and only leaves do much work)



1 new thread at each step



Divide-and-Conquer Style Code

(doesn't work in general... more on that later)

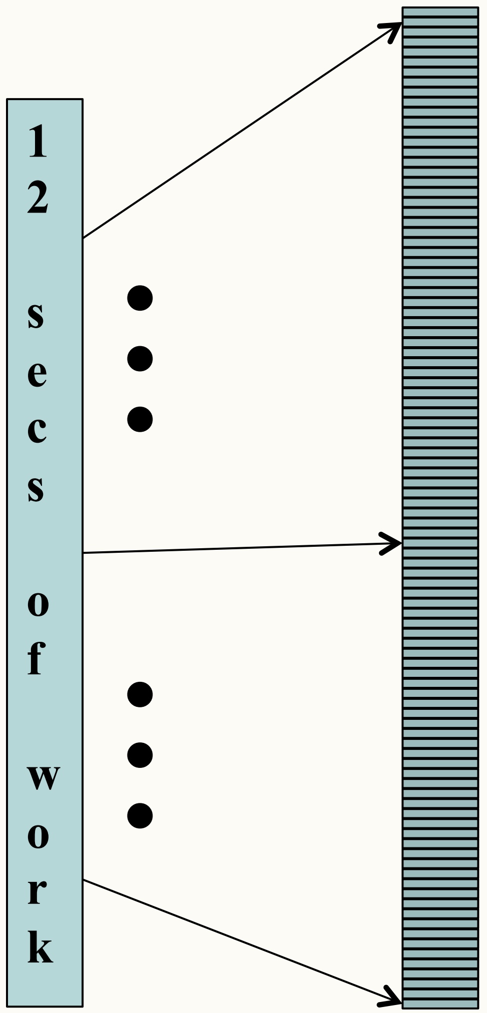
```
void cmp_helper(int * result, int array[], int lo, int hi, int target)
{
  if (len <= 1) {
    *result = count_matches(array + lo, hi-lo, target);
    return;
  }

  int left, right;
  int mid = lo + (hi-lo)/2;
  std::thread child(&cmp_helper, &left, array, lo, mid, target);
  cmp_helper(&right, array, mid, hi, target);
  child.join();

  return left + right;
}

int cm_parallel(int array[], int len, int target) {
  int result;
  cmp_helper(&result, array, 0, len, target);
  return result;
}
```

Chopping Too Fine Again



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

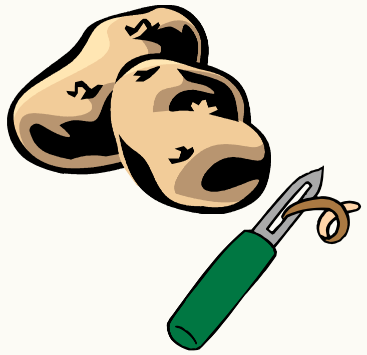
Result?

KP Duty: Peeling Potatoes, Parallelism Reminder

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?

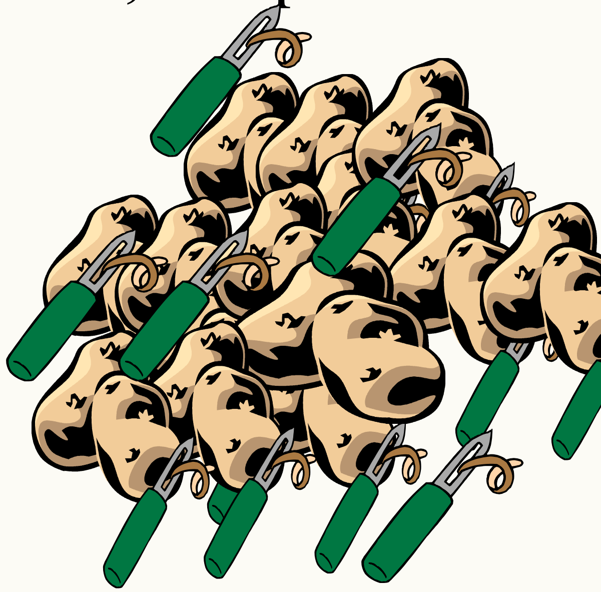


KP Duty: Peeling Potatoes, Parallelism Problem

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?



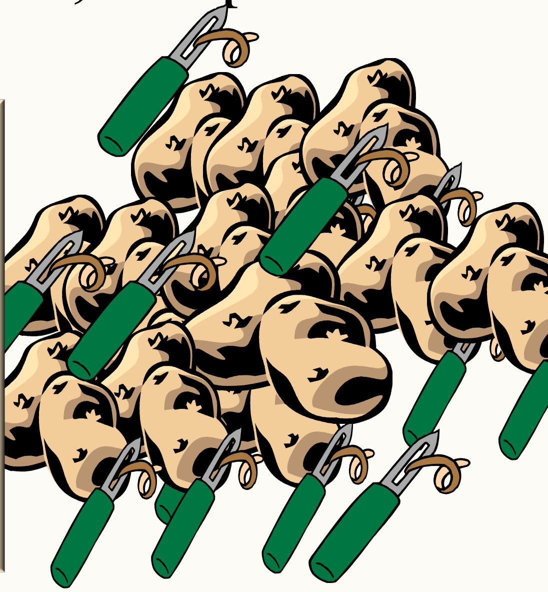
KP Duty: Peeling Potatoes, Parallelism Problem

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 much time do we spend finding places for people to work, handing out peelers, giving instructions, teaching technique, bandaging wounds, and (ack!) filling out paperwork?



Being Realistic

- Creating one thread per element is *way* too expensive.
- So, we use a library where we create "tasks" ("bite-sized" pieces of work) that the library assigns to a "reasonable" number of threads.

- But… creating one task per element *still* too expensive.
- So, we use a *sequential cutoff*, typically ~500-1000. (This is like switching from quicksort to insertion sort for small subproblems.)

Note: we're still chopping into $\Theta(n)$ pieces, just not into n pieces.

Being Realistic: Exercise

How much does a sequential cutoff help?

- Exercise: If there are ($\sim 2^{30}$) elements in the array and our cutoff is 1, about how many tasks do we create? (I.e., nodes in the tree.)

  2^{31} (2 billion)

- Exercise: If there are ($\sim 2^{30}$) elements in the array and our cutoff is 1,000 ($\sim 2^{10}$), about how many tasks do we create?

  2^{21} (2 million)

- What percentage of the tasks do we eliminate with our cutoff?

  99.9%

That Library, Finally

- C++11's threads are usually too "heavyweight" (implementation dependent).
- OpenMP version 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;
}
```

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