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

Why Parallelism?

*Why **not** Parallelism?*

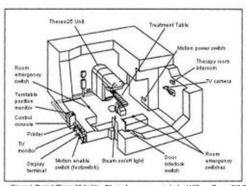


Figure 1. Typical Therac-25 facility Photo from case study by William Frey, **CC BY 3.0**

Concurrency problems were certainly not the only problem here… nonetheless, it's hard to reason correctly about programs with concurrency.

Moral: Rely as much as possible on high-quality pre-made solutions (libraries).

Learning Goals

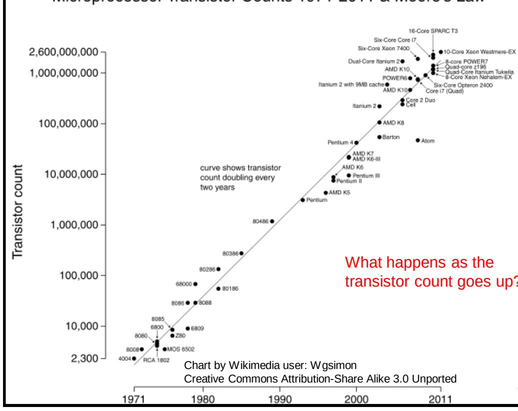
By the end of this unit, you should be able to:

- 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.)

Outline

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

Microprocessor Transistor Counts 1971-2011 & Moore's Law



(zoomed in)

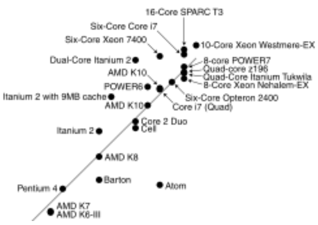
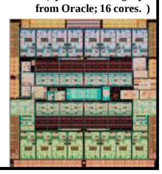


Chart by Wikimedia user: Wgsimon
Creative Commons Attribution-Share Alike 3.0 Unported

(Sparc T3 micrograph from Oracle; 16 cores.)



(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!)

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

---

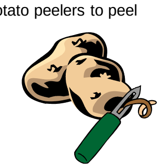
Outline

- 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?

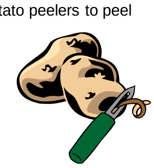


KP Duty: Peeling Potatoes, Parallelism

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!

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?

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

(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!

Concurrency Example

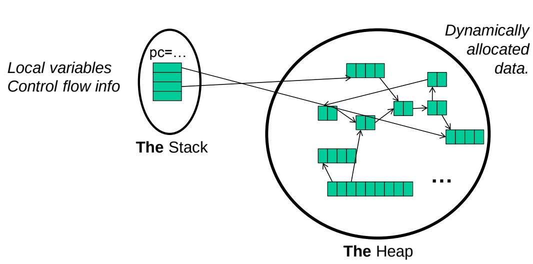
Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)

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)
   }
}
```

OLD *Memory Model*

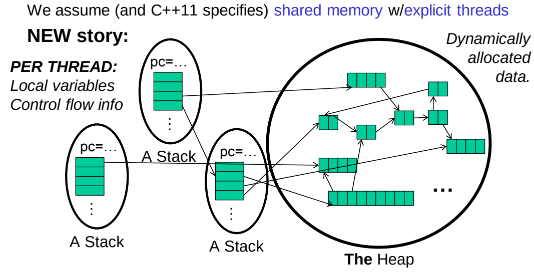


(pc = program counter, address of current instruction)

Shared Memory Model

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

NEW story:



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.

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 = 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?

Fork/Join Parallelism

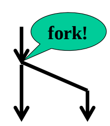
`std::thread` defines methods you could not implement on your own
- **The constructor calls its argument *in a new thread* (forks)**

Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own
- **The constructor calls its argument *in a new thread* (forks)**

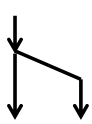
Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own
- **The constructor calls its argument *in a new thread* (forks)**



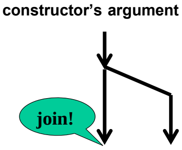
Fork/Join Parallelism

`std::thread` defines methods you could not implement on your own
- **The constructor calls its argument *in a new thread* (forks)**



Fork/Join Parallelism

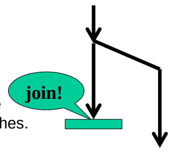
`std::thread` defines methods you could not implement on your own
- The constructor calls its argument *in a new thread* (forks)
- `join` **blocks until/unless the receiver is done executing (i.e., its constructor's argument function returns)**



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)**

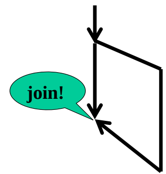


This thread is **stuck** until the other one finishes.

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)**

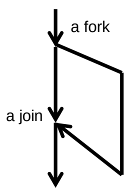


This thread could already be done (joins immediately) or could run for a long time.

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.

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();
    matches += results[d];
  }

  return matches;
}
```


Outline

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

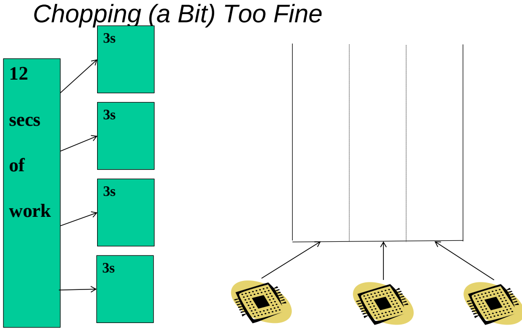
Success! Are we done?

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.)

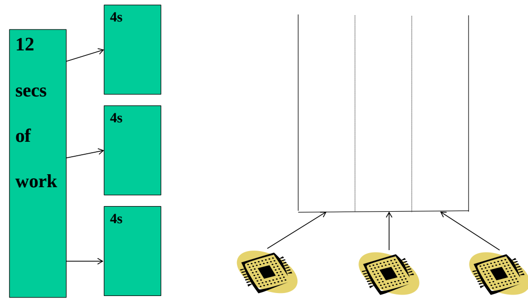
Chopping (a Bit) Too Fine



We thought there were 4 processors available.

But there's only 3. Result?

Chopping Just Right



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().)

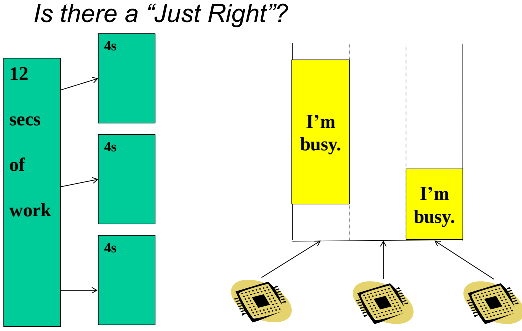
Success! Are we done?

Answer this:

- 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"?

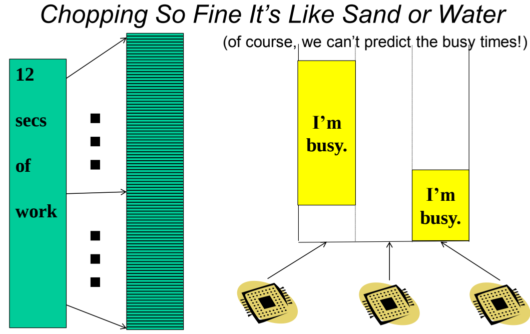


We thought there were 3 processors available.

And there are, sort of. Result?

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 a few processors. Result?

```
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;
}
```

Yes, this is silly.
We'll justify later.

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

How long does dividing up/recombining the work take?


```
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;
}
```

How long does *doing* the work take? (n == len, # of processors = ∞)

(With n threads, how much work does each one do?)


```
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;
}
```

Time $\in \Theta(n)$ with an *infinite* number of processors?
That sucks!

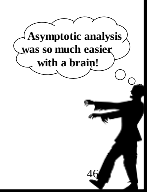
Zombies Seeking Help

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.

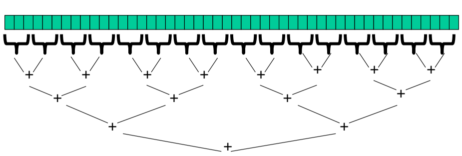
The new zombie in town has the humans line up and bites each in line, transferring the program: *Do nothing except say "Eat Brains!!"*

Analysis?

How do they do better?



A better idea



The zombie apocalypse is straightforward using divide-and-conquer

Note: the natural way to code it is to fork two tasks, join them, and get results.
But… the natural zombie way is to bite one human and then each "recurse".
(As is so often true, the zombie way is better.)

Divide-and-Conquer Style Code
(doesn't work in general... more on that later)

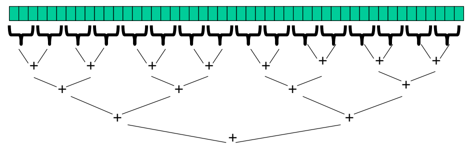
```
void cmp_helper(int * result, int array[],
                int lo, int hi, int target) {
  if (len <= 1) {
    *result = count_matches(array + lo, hi-lo, target);
    return;
  }

  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;
}
```

```
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;
}
```

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

How long does dividing up/recombining the work take? Um…?

Easier Visualization for the Analysis



How long does the tree take to run…
…with an infinite number of processors?

(n is the width of the array)


```
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;
}
```

How long does *doing* the work take? (n == len, # of processors = ∞)

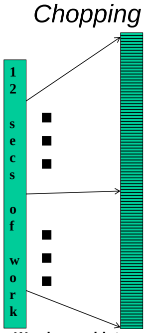
(With n threads, how much work does each one do?)


```
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;
}
```

Time ∈ Θ(lg n) with an infinite number of processors.
Exponentially faster than our Θ(n) solution! Yay!

So… why doesn't the code work?

Chopping Too Fine Again



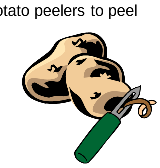
We chopped into n pieces (n == array length).

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.

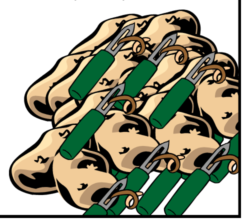
How long would it take 100 people with 100 potato peelers to peel 10,000 potatoes?



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?



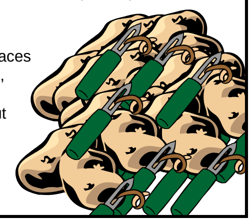
Sophomoric Parallelism and Concurrency, Lecture 1

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?



Sophomoric Parallelism and Concurrency, Lecture 1

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.

Sophomoric Parallelism and Concurrency, Lecture 1 57

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.

Sophomoric Parallelism and Concurrency, Lecture 1 58

Being realistic: Exercise

How much does a sequential cutoff help?

With 1,000,000,000 (~2^{30}) elements in the array and a cutoff of 1: **About how many tasks do we create?**

With 1,000,000,000 elements in the array and a cutoff of 16 (a *ridiculously small* cutoff): **About how many tasks do we create?**

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

Sophomoric Parallelism and Concurrency, Lecture 1 59

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!

Sophomoric Parallelism and Concurrency, Lecture 1 60

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.)

P.S. We promised we'd justify assuming # processors = ∞. Next lecture!

Outline

- History and Motivation
- Parallelism and Concurrency Intro
- Counting Matches
  - Parallelizing
  - Better, more general parallelizing
  - Bonus code and parallelism issue!

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;
}
```


Side Note: Load Imbalance

Does each "bite-sized piece of work" take the same time to run:

When counting matches?

When counting the number of prime numbers in the array?

Compare the impact of different runtimes on the "chop up perfectly by the number of processors" approach vs. "chop up super-fine".