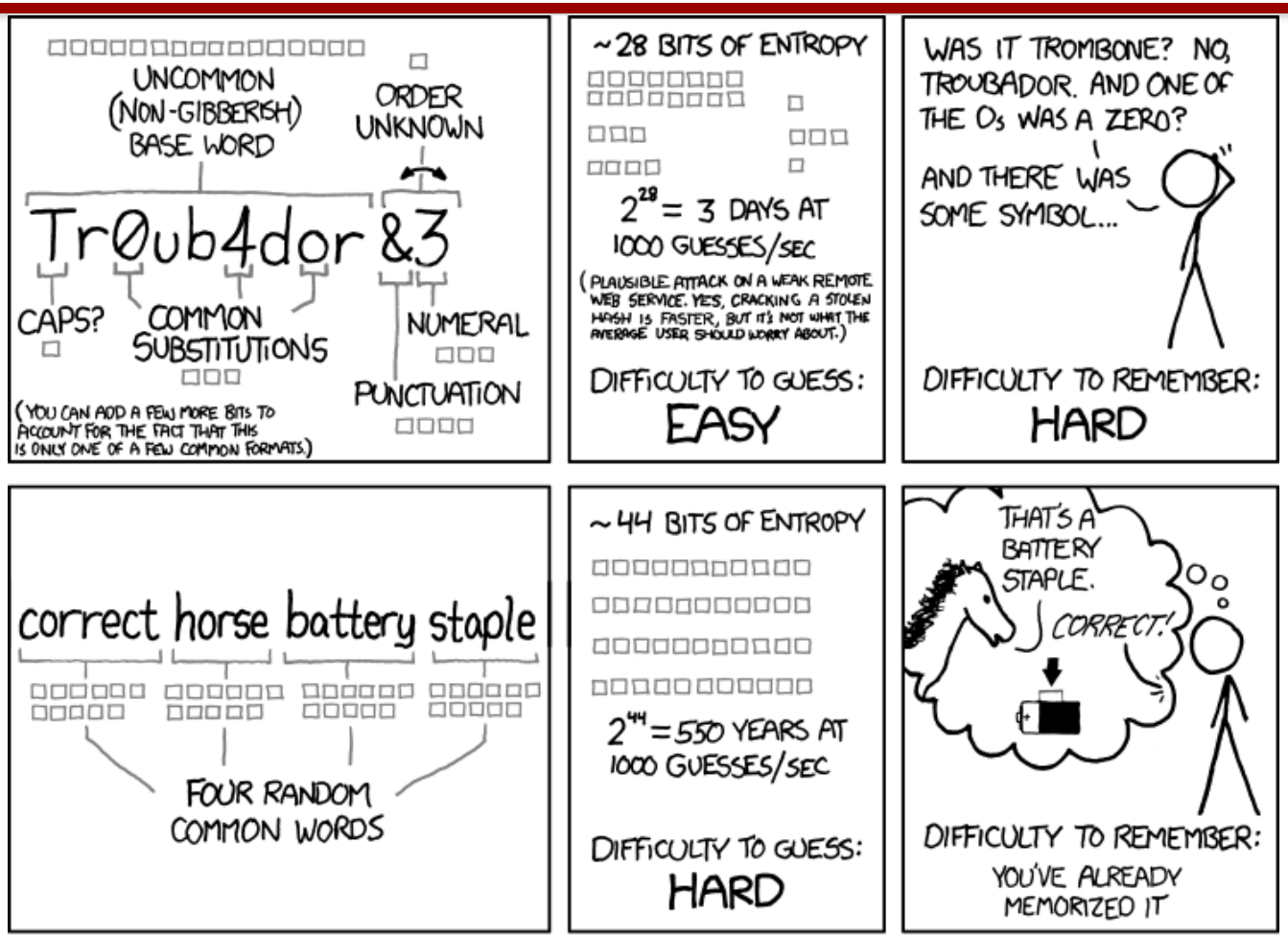
PA2 Notes; Basic Parallel Programming

ITP 435
Week 2, Lecture 1

PA 2: Passwords



PA 2: Passwords

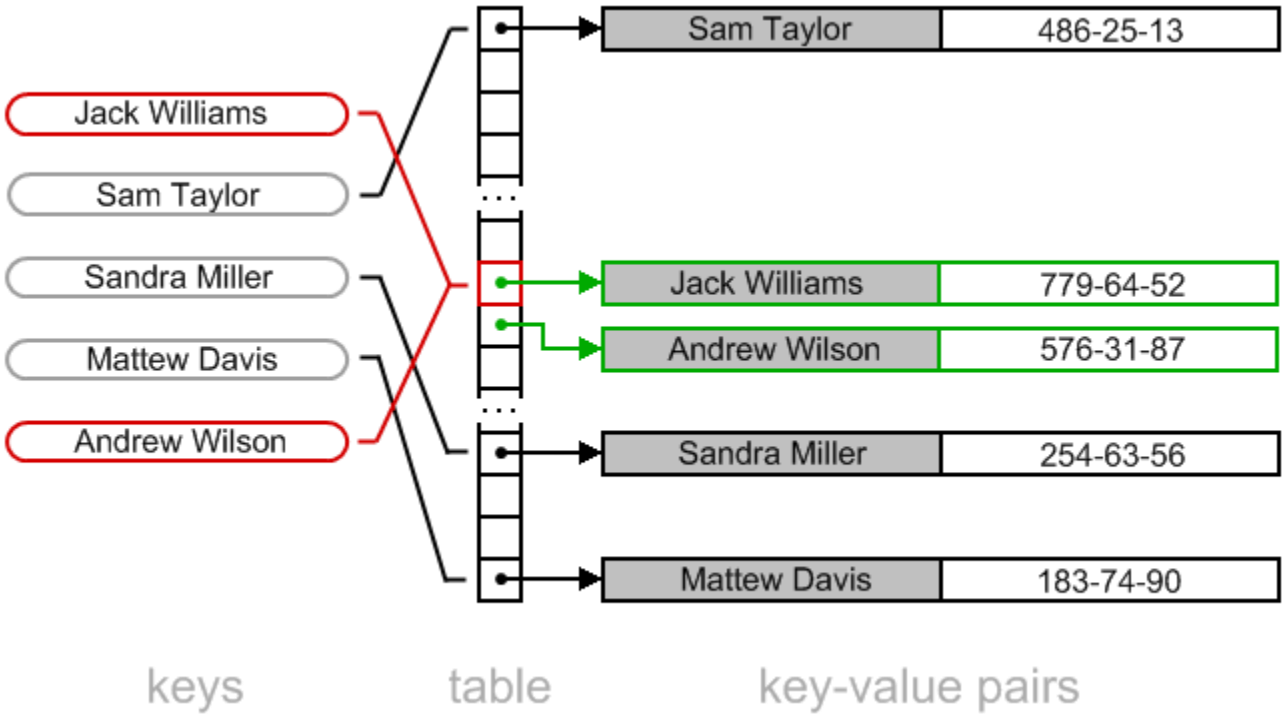
- Given some passwords encrypted (with the very insecure) sha1, figure out the original plain-text!

- Part 1 – Implement open addressed hash table (for cache performance)
- Part 2 – Implement a “dictionary” style attack
- Part 3 – Implement a “brute force” style attack

Open Addressed Hash Table

- Basic idea: If there's a collision, rather than chaining with a linked list, find the next available index and save the element there

- Here, Jack was added first and then Andrew was a collision:



Why not buckets?

- An open-addressed hash table has much better cache performance, which in many cases can make it significantly faster

- We'll talk more about cache performance next week!

HashTable declaration

```
template <typename KeyType, typename ValueType,
typename HashFuncType = std::hash<KeyType>>
class HashTable
```

- Key and Value types are templated
- Optionally, can specify a function object type for the hashing function (defaults to std::hash) – used for test cases

- We support rehashing/growing if the hash table gets full, but we don't support deleting individual elements (for simplicity)

HashTable member data

```
// Underlying hash table array
std::pair<KeyType, ValueType>* mTableData;
// Tracks whether an index is occupied
std::vector<bool> mOccupancy;
// Allocated capacity of hash table array
size_t mCapacity;
// Number of elements in HashTable
size_t mSize;
// Hash function
HashFuncType mHashFunc;
```

- *You should not add anything else!!*

HashTable::Insert

```
bool Insert(const KeyType& key, const ValueType& value)
```

- Try index at hashFuncResult % capacity
- If not occupied, insert, otherwise find the next available index
- If you reach the end and still no space, wrap around to index 0
- Insert only returns false if the key is already in the table. Otherwise, it returns true

- ***If the table is full when you try to insert, Rehash the table with double the current capacity

HashTable::Rehash

```
bool Rehash(size_t newCapacity)
```

- If the new capacity is less than the number of elements in the table, returns false
- Otherwise will make a new backing array of the requested capacity and hash everything into the new table

HashTable::Find

```
ValueType* Find(const KeyType& key) const
```

- Finds the element, if it exists, using the same “linear probing” technique from Insert
- ***If you encounter an unoccupied index, then it’s not in the table***
- Don’t get stuck in an infinite loop
- Returns pointer to value if found, nullptr otherwise

HashTable::ForEach

```
void ForEach(std::function<void(const KeyType& key,
                                ValueType& value)> func)
```

- Executes given function on each element in the table – more on this soon
- Removes need for iterators mostly

In-class Week 2 Lecture 2

- Let's trace through hash table find and insert!

HashTable Rule of 5 Functions

- You've gotta implement them all!

- Check previous lecture's notes

HashTable order to implement

1. Find
2. Insert
3. ForEach
4. Rule of 5 functions

And then hopefully you pass all the test cases

Dictionary Attack

Dictionary Attack

1. Take a list of common passwords
2. Encrypt them using sha1
3. Store in the hash table (key is encrypted string, value is original text)
4. Then lookup each “unknown” sha1 encrypted password and see if it’s in the dictionary

Passphrase Brute Force

- Given a list of words, generate all possible 4 word permutations like:

```
correcthorsebatterystaple
handbusinessworkcorrect
jobfactbookmoney
worldworldplacenight
nightbookgovernmentissue
```

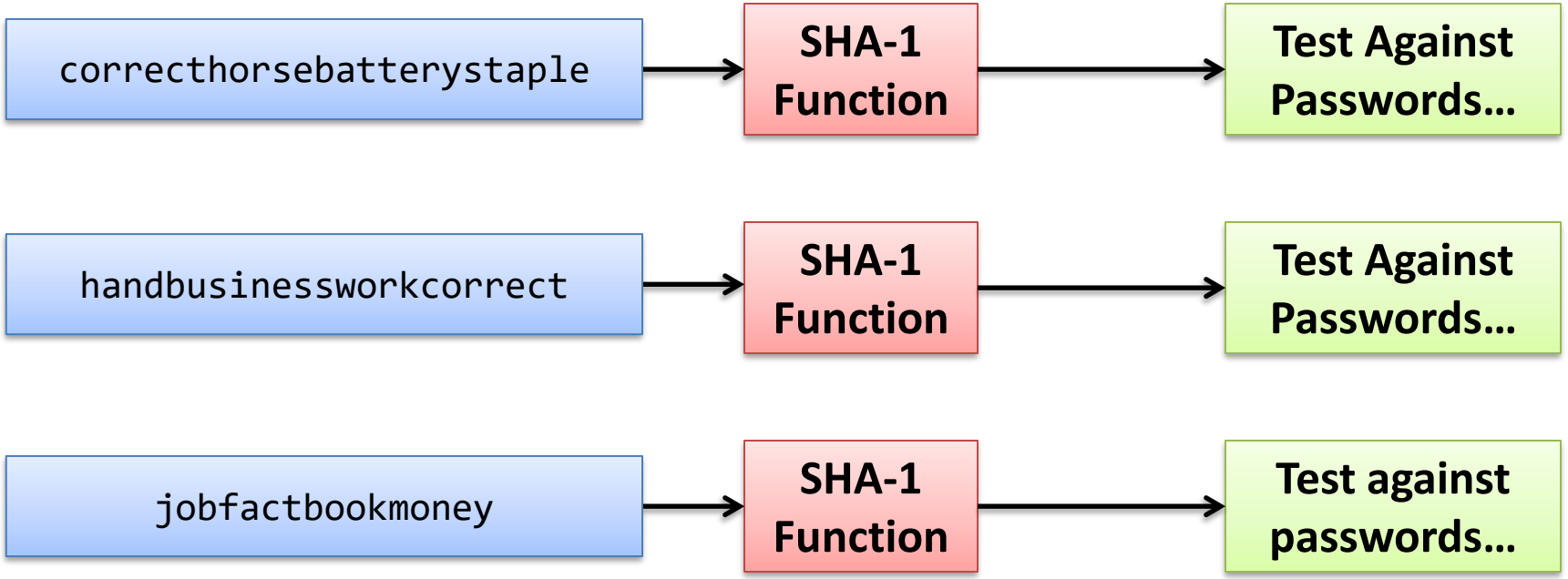
- Use sha1 on each permutation and see if it's in the hash table of unknown passwords

Number of Phrases

- The biggest input file contains 50 possible words, so…
- 50^4 = 6,250,000
- It's not really a crazy amount, but increasing the number of words significantly increases this

Dependencies?

- Are there any dependencies between all the different permutations?



A Serial Implementation

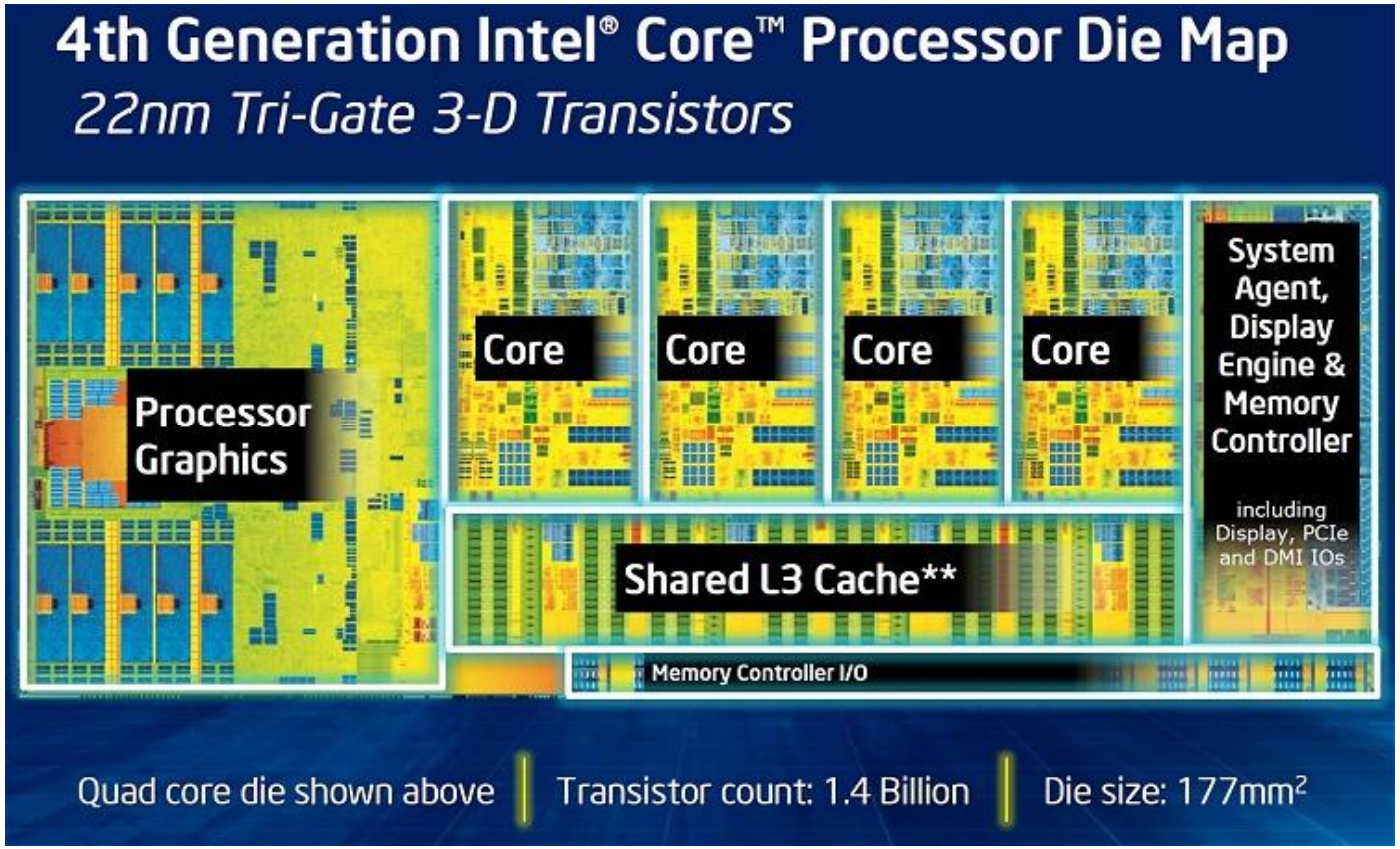
- In a ***serial*** implementation, we would just test a single permutation at a time

- On my desktop machine, the full test cases (50^4 phrases) took ~3.7s

- Serial code does not take full advantage of modern CPUs!

Multicore CPUs

- Modern CPUs are ***multi-core***, meaning they can run multiple ***threads*** (sequences of instructions) in parallel



A Parallel Approach

- If rewrite the brute force algorithm to handle multiple permutations at once…
- We can test several permutations at once, which results in a significant speedup
- The parallel implementation was first 1.6s, then 1.2s, then 0.55s, then 0.20s as I optimized it further.
- Much better than 3.7s
- *(But this can still be improved!)*

Primality Test

- A ***primality test*** returns whether or not a number is prime
- A naïve primality test (courtesy of Wikipedia):

```
bool isPrime(unsigned long n) {
    if (n <= 3) {
        return n > 1;
    } else if (n % 2 == 0 || n % 3 == 0) {
        return false;
    } else {
        for (unsigned long i = 5; i * i <= n; i += 6) {
            if (n % i == 0 || n % (i + 2) == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Primality Test in Action

- Suppose we have a file with 5 million randomly-generated unsigned integers

- For each of these integers, we want to perform a primality test, and save the result in a struct encapsulating the number:

```
struct Number {
    unsigned value;
    bool isPrime;
    Number(unsigned v) {
        value = v;
        isPrime = false;
    }
};
```

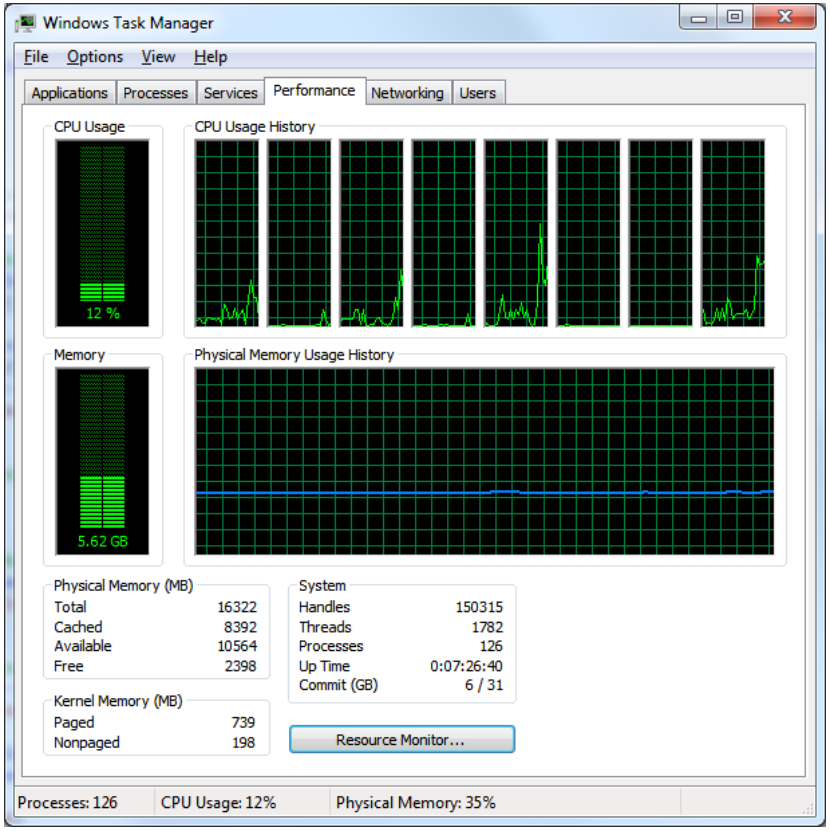
A Serial Implementation

- Here's a basic serial implementation

```
std::vector<Number> numbers;
numbers.reserve(NUMBER_COUNT);
// Populate vector with numbers from file
// ...

// Start high frequency timer (custom class I created)
Timer timer;
timer.start();

// Test each number (serial)
for (auto& n : numbers) {
   n.isPrime = isPrime(n.value);
}

// Get time elapsed
double elapsed = timer.getElapsed();
std::cout << "Took " << elapsed << " seconds" << std::endl;
```

Serial Implementation in Action

- Took ~12.5 seconds on this older machine
- Max CPU utilization was only 12%:



std::for_each

- Instead of this:

```
// Test each number (serial)
for (auto& n : numbers) {
    n.isPrime = isPrime(n.value);
}
```

- You can include <algorithm> and do:

```
std::for_each(numbers.begin(), numbers.end(),
    [](Number& n) {
        n.isPrime = isPrime(n.value);
});
```

?!?!?!?!

Lambda Expressions

Lambda Expressions – Motivation

- By default, std::sort sorts in ascending order:

```
std::vector<int> v;
v.emplace_back(10);
v.emplace_back(-10);
v.emplace_back(100);
std::sort(v.begin(), v.end());
// v = {-10, 10, 100}
```

- What if we want descending order?

A Custom Comparator!

```
std::sort(v.begin(), v.end(), [](int a, int b) {
    // Return a greater than b instead of less
    return a > b;
});
// v = {100, 10, -10}
```

- This uses a new syntax…

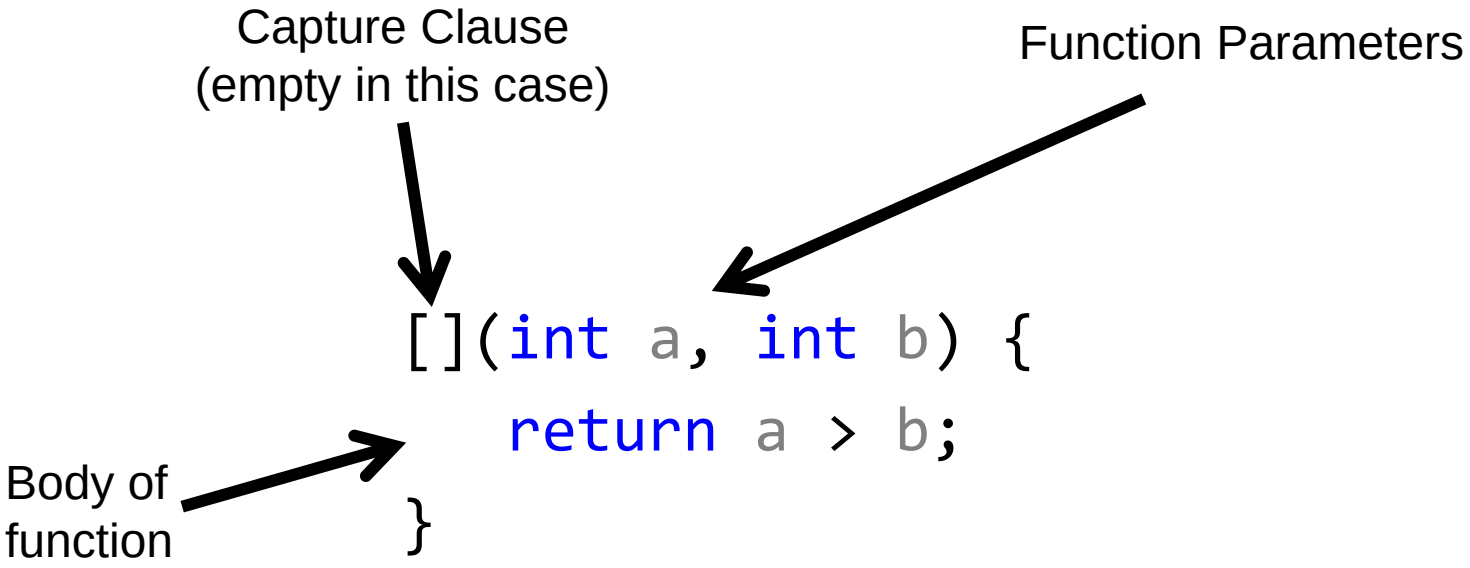
Lambda Expressions

- A ***lambda expression*** is an inline declaration of a function and implementation

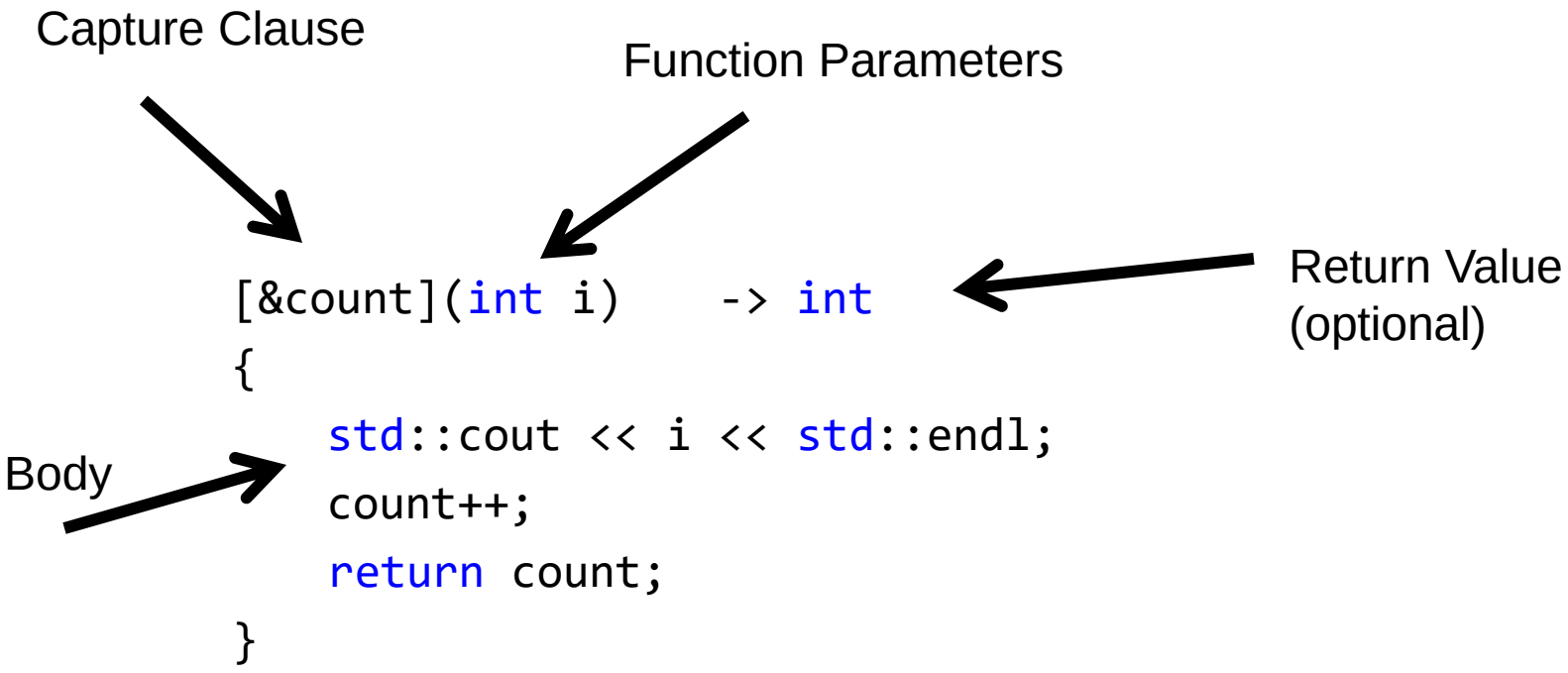
```
[](int a, int b) {
    // Return a greater than b instead of less
    return a > b;
}
```

- In other languages, this may be called anonymous functions (like in JavaScript)
- Let's explain this crazy syntax!

Lambda Expression Syntax



Lambda Expression Syntax, Another Example



Capture Clause

- This is used to “capture” variables that exist outside the lambda expression, and use them inside the lambda
- Variables can be captured by value or reference:

```
// Capture ALL local variables by value (not recommended)
[=]
// Capture ALL local variables by reference (not recommended)
[&]
// Capture x by value and y by reference
[x, &y]
// Capture count by reference, and all other locals by value
[=, &count]
// Capture this by value (can’t be captured by reference)
// If you want to use any member functions or variables in the
// lambda, you have to capture this.
[this]
```

Lambdas stored in variables

- You're allowed to store lambdas in a variable
- Instead of figuring out the type, use auto
- Our previous example could be:

```
auto greater = [](int a, int b) {
    return a > b;
};
std::sort(v.begin(), v.end(), greater);
// v = {100, 10, -10}
```

Specifying the Type of a Lambda

- If you need to specify the type, use the std::function template class (in the <functional> header):

```
std::function<bool(int, int)> greater =
    [](int a, int b) {
        return a > b;
};
```

- (This says the lambda returns a bool and takes in two ints as parameters)

- To call it, use like a function:

```
bool result = greater(5, 10);
```

Parallel Algorithms

std::for_each

- Instead of this:

```
// Test each number (serial)
for (auto& n : numbers) {
    n.isPrime = isPrime(n.value);
}
```

- You can include <algorithm> and do:

```
std::for_each(numbers.begin(),numbers.end(),
    [](Number& n) {
        n.isPrime = isPrime(n.value);
});
```

Parallel Algorithms

- As of C++17, STL algorithms now support parallelization

Table 1 — Table of parallel algorithms

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

Using Parallel Algorithms

- You need to include `<algorithm>` like usual, but you also need to include `<execution>`

- Supported STL algorithms now have an overloaded version that includes an execution policy as the first parameter

- Old version (still works if you want to use it):

```
std::for_each(v.begin(), v.end(), ...);
```

- New version (supports execution policy):

```
std::for_each(std::execution::par, v.begin(),
v.end(), ...);
```

Clang/Mac Important Note

- Clang's STL implementation on Mac doesn't support parallel algorithms, so we're using someone's partial implementation of it (which works for us)
- But requires this janky code…

```
#if defined(__APPLE__)
// On Apple platforms, import a 3rd-party parallel implementation
#define PSTLD_HEADER_ONLY
#define PSTLD_HACK_INTO_STD
#include "pstld.h"
#else
// On other platforms, STL works!
#include <algorithm>
#include <execution>
#endif
```

Execution Policies

- `std::execution::seq`
  - This is just like algorithms behaved before parallel algorithms
  - The algorithm runs on a single thread in sequential order from the start to end of the range
- `std::execution::par`
  - The work can be split up between multiple threads
  - However, each thread will work on a contiguous part of the sequence and in sequential order for its part
- `std::execution::par_unseq`
  - The work can be parallel and also interleaved with no guaranteed ordering between elements

Parallelization is not guaranteed

- Keep in mind that the execution policies are ***suggestions***:

"If the implementation cannot parallelize or vectorize (e.g. due to lack of resources), all standard execution policies can fall back to sequential execution."

- *Also note that not every compiler currently supports parallel algorithms (for example, clang's STL implementation does not)*

std::for_each, parallel

- Instead of this:

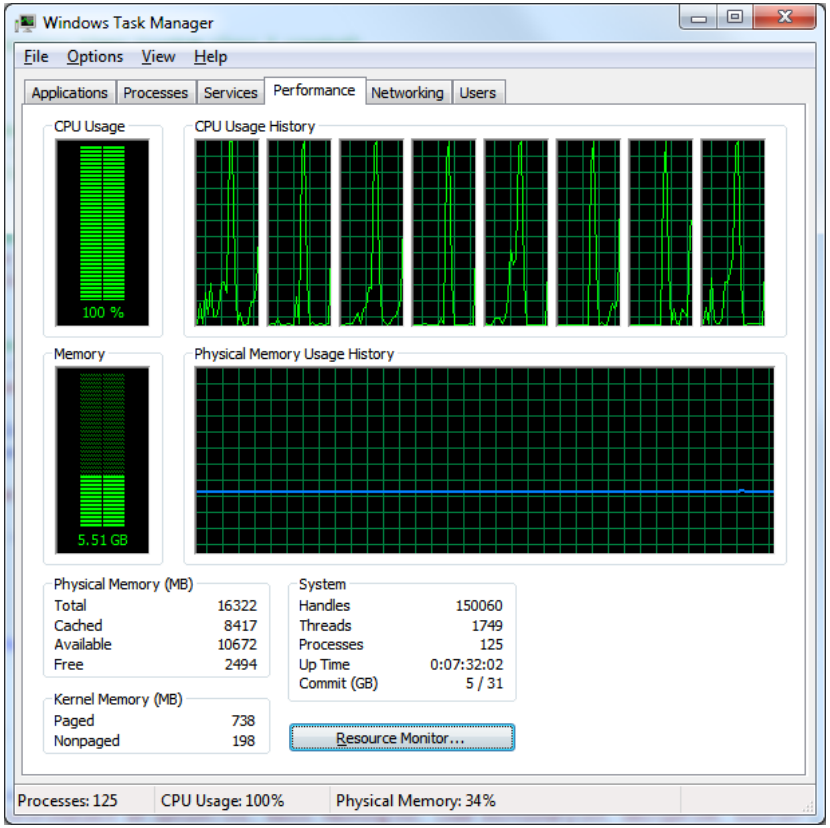
```
std::for_each(numbers.begin(),numbers.end(),
    [](Number& n) {
        n.isPrime = isPrime(n.value);
});
```

- Add the parallel execution policy!

```
std::for_each(std::execution::par, numbers.begin(),
    numbers.end(),
    [](Number& n) {
        n.isPrime = isPrime(n.value);
});
```

std::for_each, parallel, in action

- Takes only ~2.27 seconds, a **5.5x** improvement
- We also hit max CPU utilization!



Embarrassingly Parallel

- The problem we just solved is considered an ***embarrassingly parallel problem*** – a problem for which little or no effort is required to separate the problem into parallel tasks

- Some other examples:
  - Brute forcing passwords (as we do in PA2)
  - Rendering independent frames (computer graphics)
  - Simulating independent particles
  - Many different types of fractal/noise algorithms

Another Example

- Suppose you have this double for loop:

```
for (int i = 0; i < 20; i++) {
    std::string temp; // some temp data...
    for (int j = 0; j < 20; j++) {
        // Do stuff with 2D vector v...
        v[i][j] = /*...*/;
    }
}
```

- How to convert it to a parallel `for_each`?

- ***Idea***: Convert the outermost loop to a `for_each` but leave the rest the same

Naïve idea: vector of indices

- Make a vector of the values of `i` and `for_each` over that:

```
// Construct a vector of 20 elements
std::vector<int> indices(20);
// std::iota will put the values from 0 to n into the vector
std::iota(indices.begin(), indices.end(), 0);

// Now do the for_each
std::for_each(std::execution::par, indices.begin(), indices.end(),
   // Capture v by reference so we can use it inside the lambda!
   [&v](int i) {
      std::string temp; // some temp data...
      for (int j = 0; j < 20; j++) {
         // Do stuff with 2D vector v...
         v[i][j] = /*...*/;
      }
});
```

Better Idea: Subdivide into ranges

```
// Construct a vector of pairs of the ranges
std::vector<std::pair<int, int>> ranges{
   {0, 5}, {5, 10}, {10, 15}, {15, 20}
};

// for_each over the pairs
std::for_each(std::execution::par, ranges.begin(), ranges.end(),
   [&v](const std::pair<int, int>& r) {
      for (int i = r.first; i < r.second; ++i) {
         std::string temp; // some temp data...
         for (int j = 0; j < 20; j++) {
            // Do stuff with 2D vector v...
            v[i][j] = /*...*/;
         }
      }
});
```

How many pieces to split into?

- Rather than hard-coding the number of splits, it's best to think in terms of how many cores you have, and programmatically split based on that

- `std::thread::hardware_concurrency()` in `<thread>` tells you the number of cores

- You can use something like:

```
unsigned numSplits = std::max(4u, std::thread::hardware_concurrency());
```

- (So ***always*** split into at least 4 pieces, but into more pieces if you have more cores)

Is parallel always better?

- There's a cost associated with parallelization

- Too many calls can be significantly slower!

Grain Size

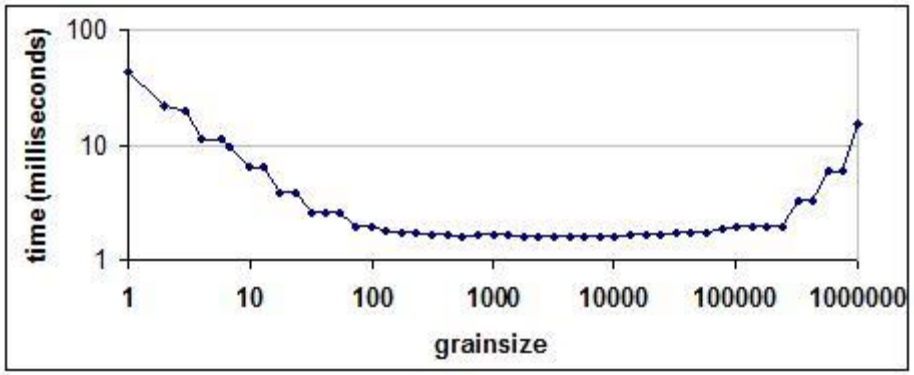
- A ***grain size*** determines the smallest size piece the workload is broken down into

- If the grain size is too small (as in the first parallel Fibonacci implementation), then the parallel solution will be slower than the serial one

Grain Size

- A ***grain size*** determines the smallest size piece the workload is broken down into

- Too small of a grain size and overhead dominates, too big and you don't take advantage of parallelization



- So, we need to be smart about how we split up work

Common Parallel Pitfall

- Sharing data between parallel operations is okay if it's read-only

- However, if you ***write*** to shared data, you have to be very careful

```
std::vector<int> test = { 1, 1, 2, 3, 5 };
std::vector<int> copy;

// Let's use parallel_for_each to copy!
std::for_each(std::execution::par, test.begin(), test.end(),
   [&](int i) {
      copy.push_back(i); // This is bad...
});

for (auto i : copy) {
   std::cout << i << std::endl;
}
```

Result of Parallel Copy

- Surprisingly, it doesn't crash, but we copy out of order:

1

1

5

2

3

STL Collections and Parallel Programming

- You should assume that adding/removing elements from an STL collection ***is unsafe in a parallel algorithm*** – they aren't designed for this

- Reading specific elements should be okay, as long as no one else is adding/removing elements at the same time

- Writing to specific elements may be okay, depending on how it's done

One Last Look at Execution Policies…

Execution Policies

- `std::execution::seq`
  - This is just like algorithms behaved before parallel algorithms
  - The algorithm runs on a single thread in sequential order from the start to end of the range
- `std::execution::par`
  - The work can be split up between multiple threads
  - However, each thread will work on a contiguous part of the sequence and in sequential order for its part
- *`std::execution::par_unseq`*
  - The work can be parallel and also interleaved with no guaranteed ordering between elements

Example

- Suppose we have two input vectors
- We can transform it into one output vector – our transform will multiply the pairs of elements together

- Old (sequential) way:

```
std::transform(
   inputA.begin(), inputA.end(), // Begin/end of 1st input range
   inputB.begin(), // Begin of 2nd input range
   destSeq.begin(), // Begin of destination range
   [](float x, float y) { return x * y; } // Lambda
);
```

How does sequential execute?

- Suppose there's 100 elements.
- This will execute in sequence with no parallelization:

```
destSeq[0] = inputA[0] * inputB[0]
destSeq[1] = inputA[1] * inputB[1]
destSeq[2] = inputA[2] * inputB[2]
destSeq[3] = inputA[3] * inputB[3]
destSeq[4] = inputA[4] * inputB[4]
destSeq[5] = inputA[5] * inputB[5]
destSeq[6] = inputA[6] * inputB[6]
destSeq[7] = inputA[7] * inputB[7]
...
destSeq[99] = inputA[99] * inputB[99]
```

Parallel Execution Policy

- The transform call looks the same except for specifying the policy:

```
std::transform(
    std::execution::par, // Parallel execution policy
    inputA.begin(), inputA.end(), // Begin/end of 1st input range
    inputB.begin(), // Begin of 2nd input range
    destPar.begin(), // Begin of destination range
    [](float x, float y) { return x * y; } // Lambda
);
```

How does parallel execute?

- ***Potentially*** it will split the range into multiple threads
- (For example, if it's two threads and there's 100 elements)

Thread 0	Thread 1

`destSeq[0] = inputA[0] * inputB[0]`
`destSeq[1] = inputA[1] * inputB[1]`
`destSeq[2] = inputA[2] * inputB[2]`
`destSeq[3] = inputA[3] * inputB[3]`
`destSeq[4] = inputA[4] * inputB[4]`
`destSeq[5] = inputA[5] * inputB[5]`
`destSeq[6] = inputA[6] * inputB[6]`
`destSeq[7] = inputA[7] * inputB[7]`
`...`
`destSeq[49] = inputA[49] * inputB[49]`
	`destSeq[50] = inputA[50] * inputB[50]`
	`destSeq[51] = inputA[51] * inputB[51]`
	`destSeq[52] = inputA[52] * inputB[52]`
	`destSeq[53] = inputA[53] * inputB[53]`
	`destSeq[54] = inputA[54] * inputB[54]`
	`destSeq[55] = inputA[55] * inputB[55]`
	`destSeq[56] = inputA[56] * inputB[56]`
	`destSeq[57] = inputA[57] * inputB[57]`
	`...`
	`destSeq[99] = inputA[99] * inputB[99]`

Parallel/Unsequenced Execution Policy

- Just has a different policy:

```
std::transform(
    std::execution::par_unseq, // Parallel/unsequenced policy
    inputA.begin(), inputA.end(), // Begin/end of 1st input range
    inputB.begin(), // Begin of 2nd input range
    destPairUnseq.begin(), // Begin of destination range
    [](float x, float y) { return x * y; } // Lambda
);
```

How does parallel_unsequenced execute?

- ***Potentially*** could interleave the operations and ***potentially*** could vectorize operations

Thread 0	Thread 1

`destSeq[0] = inputA[0] * inputB[0]`
`destSeq[2] = inputA[2] * inputB[2]`
`destSeq[4] = inputA[4] * inputB[4]`
`destSeq[6] = inputA[6] * inputB[6]`
`...`
`destSeq[98] = inputA[98] * inputB[98]`
	`destSeq[1] = inputA[1] * inputB[1]`
	`destSeq[3] = inputA[3] * inputB[3]`
	`destSeq[5] = inputA[5] * inputB[5]`
	`destSeq[7] = inputA[7] * inputB[7]`
	`...`
	`destSeq[99] = inputA[99] * inputB[99]`

Parallelization is not guaranteed

- Keep in mind that the execution policies are ***suggestions***:

"If the implementation cannot parallelize or vectorize (e.g. due to lack of resources), all standard execution policies can fall back to sequential execution."