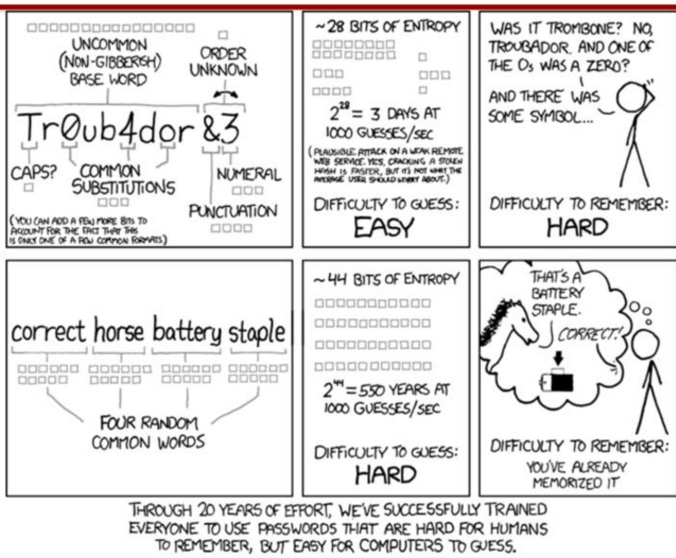
Basic Parallel Programming; Intel TBB

ITP 435 – Spring 2016
Week 3, Lecture 1
Lecturer: Sanjay Madhav

USC Viterbi School of Engineering

University of Southern California

PA 2: Passwords



USC Viterbi
School of Engineering

University of Southern California

From xkcd: http://xkcd.com/936/

PA 2: Passwords

- pass.txt contains a list of 5,000 passwords which were encoded using SHA-1
- Most passwords are in the dictionary (d8.txt)

Steps:

1. Hash all the passwords in d8.txt
2. Compare those hashed passwords against the ones in pass.txt
3. If the hash matches one in the dictionary, you've solved it
4. If the hash isn't in the dictionary, save it in an unsolved lists
5. Once the dictionary lookup is done, any remaining passwords will be brute forced.

Brute Forcing

- Let's say we're brute forcing 4-digit PINs
- The range of values are from 0000 to 9999 (10,000 permutation)

- How would you implement this on a computer where an `int` can only store values from 0 to 9?

Counting Machine

- We would need an array of 4 ints, one for each digit.
- Initialize each digit to 0, test it, then start adding 1 to the "ones" digit

TEST:	**0**	**0**	**0**	**0**
				+1
TEST:	**0**	**0**	**0**	**1**

Counting Machine, cont'd

- Once we get to 0009, the next increment will cause a carry to propagate:

TEST:	**0**	**0**	**0**	**9**
				+1
TEST:	**0**	**0**	**1**	**0**

USC Viterbi School of Engineering

University of Southern California

Counting Machine (Arbitrary Base)

- If you can successfully implement a counting machine for base 10, there's no reason why you couldn't do one for base 36:

TEST:	0	0	0	35
				+1
TEST:	0	0	1	0

Base 36

- Why base 36?
- Our passwords are limited to lower case letters and numerals
- There's 26 letters and 10 numerals.

- So, if we want to brute force passwords with the above criteria, we would need a base 36 counting machine
- Then we just need a conversion from numeral to character, eg:

```
0 = 'a'
1 = 'b'
…
25 = 'z'
…
35 = '9'
```

USC Viterbi School of Engineering

University of Southern California

Note: don't use some massive if/else chain or switch for this. Take advantage of the extra information you know about ASCII

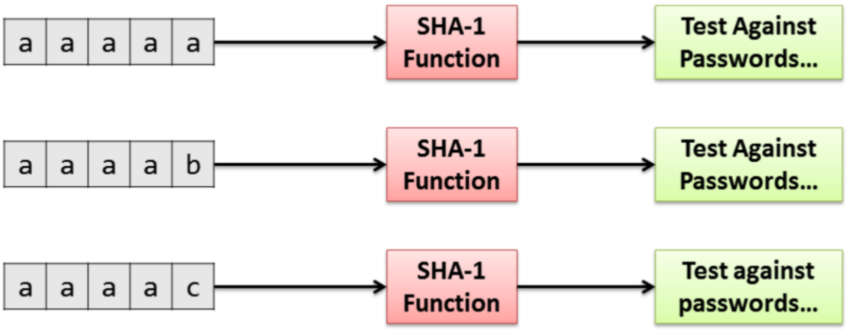
5 Characters

- **Q:** How many combinations to test if there are 5 characters?

- **A:** 36^5 = 60,466,176

Dependencies?

- Are there any dependencies between all the 60 million+ permutations?



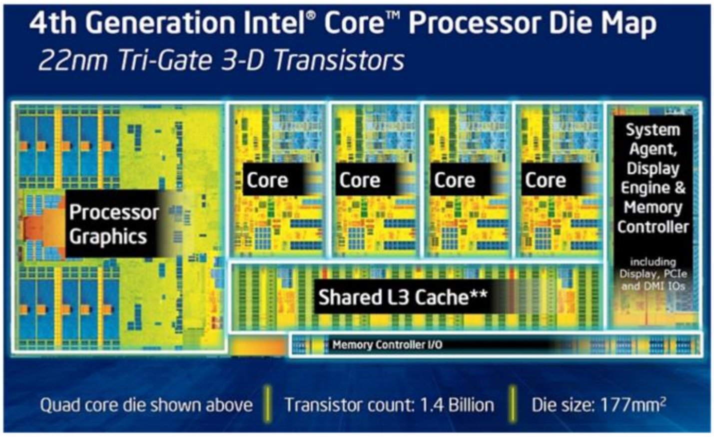
- (In other words, does "aaaac" need to know anything about the SHA-1 hash of "aaaab" or any other permutation?)

A Serial Implementation

- In a ***serial*** implementation, we would just test one permutation after another. This could be just one big loop.
- On my desktop machine, the serial implementation of 5 character brute-force takes ~16 seconds
- Serial code does not take full advantage of modern CPUs!

Multicore CPUs

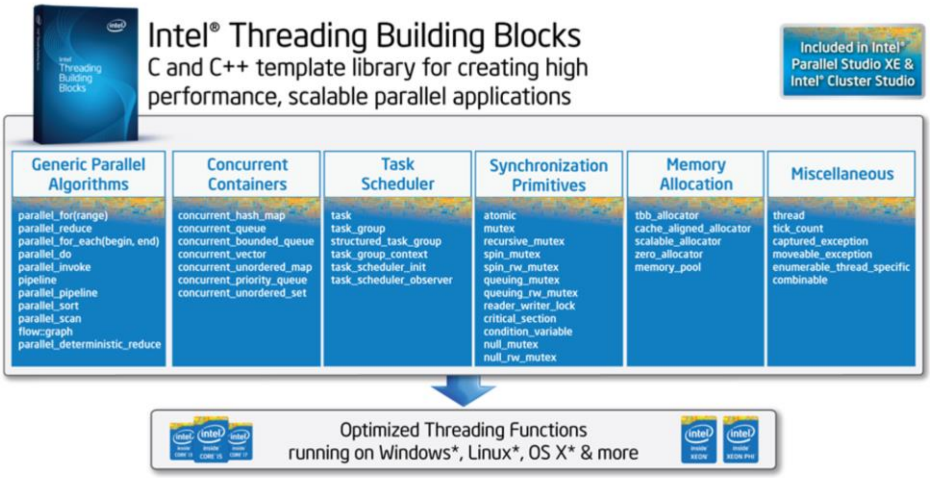
- Modern CPUs are ***multi-core***, meaning they can run multiple ***threads*** (sequences of instructions) in parallel



A Parallel Approach

- If rewrite the brute force algorithm to handle multiple permutations at once…
- We can test ~4 permutations at once, which results in a significant speedup
- On my desktop the time goes from ~16s to 5s…almost a 4x improvement!
- *(But this can still be improved!)*

Intel Thread Building Blocks



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;
   }
}
```

USC Viterbi
School of Engineering

University of Southern California

http://en.wikipedia.org/wiki/Primality_test

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;
   }
};
```

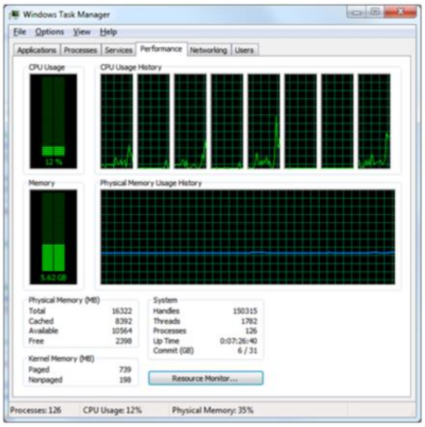
A Serial Implementation

- 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
- Max CPU utilization was only 12%:



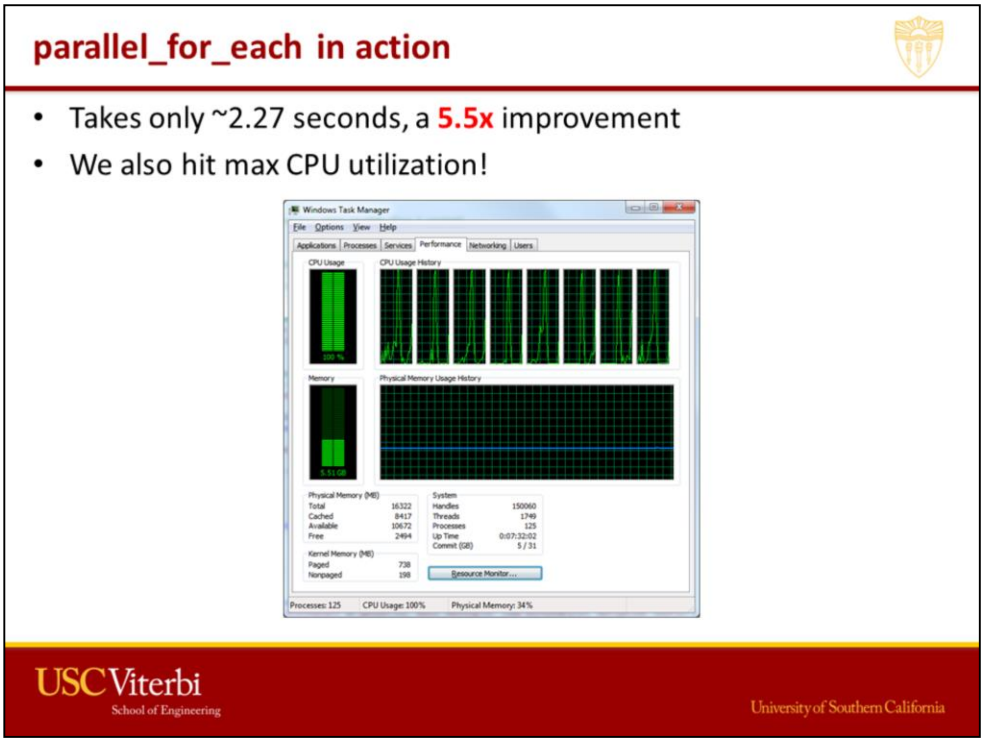
parallel_for_each

- `tbb::parallel_for_each` is very similar to the STL `for_each`, except it distributes the work across all of the cores
- Include `<tbb/parallel_for_each.h>`
- Takes three parameters:
  - Beginning of range
  - End of range
  - Lambda expression to apply to each element in the range

parallel_for_each Code

- We can't use a range-based for, but other than that the code is not that much different:

```
// Test each number (parallel)
tbb::parallel_for_each(numbers.begin(),numbers.end(),
   [](Number& n) {
      n.isPrime = isPrime(n.value);
});
```



So we got a huge performance gain without really having to do much of anything

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

USC Viterbi School of Engineering

University of Southern California

Unfortunately, not all problems can be parallelized this easily!

parallel_invoke

- `tbb::parallel_invoke` can be used to execute 2-9 lambda expressions concurrently – the function will return once all the lambdas finish their work
- Include `<tbb/parallel_invoke.h>`
- Example:

```
tbb::parallel_invoke(
   [this] { BruteForceHelper(unsolvedPass, 0, 3); },
   [this] { BruteForceHelper(unsolvedPass, 4, 7); },
   [this] { BruteForceHelper(unsolvedPass, 8, 11); },
   [this] { BruteForceHelper(unsolvedPass, 12, 15); },
   [this] { BruteForceHelper(unsolvedPass, 16, 19); },
   [this] { BruteForceHelper(unsolvedPass, 20, 23); },
   [this] { BruteForceHelper(unsolvedPass, 24, 27); },
   [this] { BruteForceHelper(unsolvedPass, 28, 31); },
   [this] { BruteForceHelper(unsolvedPass, 32, 35); }
);
```

In this case, I use parallel_invoke to split up the brute force search space between multiple tasks, which is how I achieve my fast times on brute forcing

Is parallel_invoke always better?

- There's a cost associated with running functions via `parallel_invoke`

- Too many calls can be significantly slower!

Serial Fibonacci Numbers

- Calculating a Fibonacci number using recursion (serial):

```
unsigned serialFib(unsigned n) {
   if (n < 2) {
      return n;
   } else {
      return serialFib(n - 2) + serialFib(n - 1);
   }
}
```

Parallel Fibonacci Numbers

- Calculating a Fibonacci number using parallel_invoke:

```
void parallelFib(unsigned n, unsigned& sum) {
   if (n < 2) {
      sum = n;
   } else {
      unsigned x, y;
      tbb::parallel_invoke(
         [&x,&n]() { parallelFib(n - 2, x); },
         [&y,&n]() { parallelFib(n - 1, y); }
      );
      sum = x + y;
   }
}
```

USC Viterbi School of Engineering

University of Southern California

Performance Comparsion

- Calculating the 40th Fibonacci number…
- `serialFib(40)` takes 0.33 seconds
- `parallelFib(40, result)` takes 5.6 seconds ☹

USC Viterbi
School of Engineering

University of Southern California

The parallel one is significantly slower because of the overhead in calling parallel_invoke so many times

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
- Let's rewrite the parallel Fibonacci computation so it only splits up for n > 20…

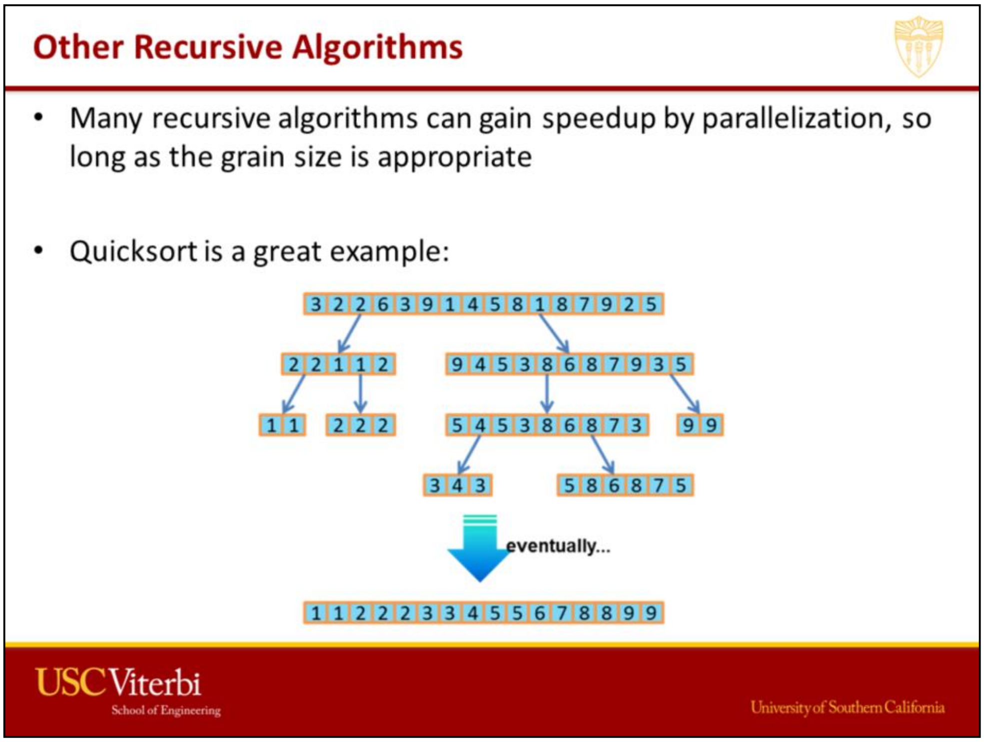
Parallel Fibonacci Numbers, v2.0

```
void parallelFib(unsigned n, unsigned& sum) {
   if (n < 2) {
      sum = n;
   } else if (n < 20) {
      // For small n values, just call serialFib
      sum = serialFib(n);
   } else {
      unsigned x, y;
      tbb::parallel_invoke(
         [&x,&n]() { parallelFib(n - 2, x); },
         [&y,&n]() { parallelFib(n - 1, y); }
      );
      sum = x + y;
   }
}
```

Parallel Fibonacci v2.0 Performance

- By making the grain size larger, the algorithm now takes only 0.07 seconds!
- **4.7x** faster than the serial version (and **80x** faster than the original parallel version)

Moral: grain size is important



TBB provides this in tbb::parallel_sort

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!
tbb::parallel_for_each(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 operator*** – 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

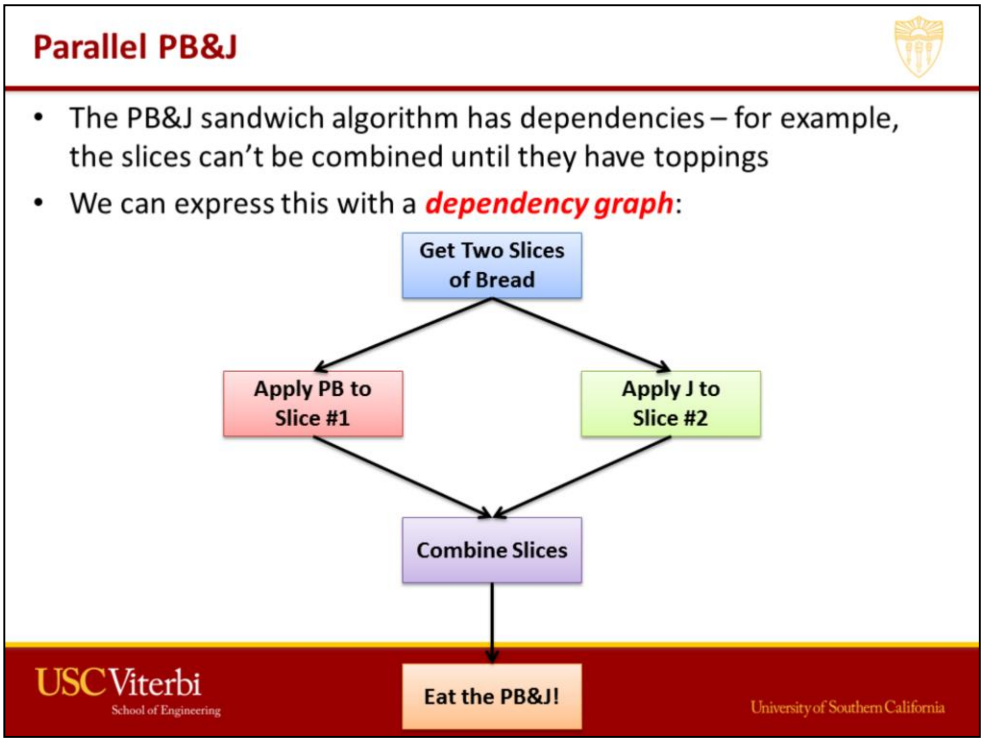
Concurrent Collections

- TBB provides a handful of concurrent collections including:
  - concurrent_unordered_map/set
  - concurrent_vector
  - concurrent_queue
  - concurrent_priority_queue
- You can safely add/remove elements in parallel programs if you use these collections
- By necessity, concurrent collections ***will be slower*** than their non-concurrent counterparts.

Let's Make a PB&J

A serial algorithm would be:

1. Get two slices of bread
2. Apply peanut butter to one slice
3. Apply jelly to the other slice
4. Combine the slices
5. Eat the PB&J!



The edges represent the dependencies – so for example you can't apply peanut butter or jelly until you have two slices of bread, and you can't combine the slices until those steps are done

Flow Graphs in TBB

- TBB has a very powerful flow graph system, which allows you to express a dependency graph in code
- You can then execute the calculations represented by the graph!
- Include `<tbb/flow_graph.h>`
- Step 1:

```
// So I don't have to type tbb::flow over and over
using namespace tbb::flow;
// Create a graph
graph g;
```

Create a Node

- The most basic type of node is a continue_node, which will continue once it receives a continue_msg from all of its dependencies

- The constructor for a continue_node takes:
  - The graph it's part of
  - A lambda expression to execute once the node has satisfied its dependencies

```
// Node for get bread
continue_node<continue_msg> getBread(g,
   [](const continue_msg&) {
      std::cout << "Getting bread...\n";
});
```

USC Viterbi School of Engineering

University of Southern California

Peanut Butter Node

```
// Node for apply peanut butter
continue_node<continue_msg> applyPB(g,
   [](const continue_msg&) {
      std::cout << "Applying peanut butter...\n";
});
```

Creating Edges

- You use make_edge to create an edge from a node to the dependent node:

```
// Create edge from getBread to applyPB --
// This creates the dependency
make_edge(getBread, applyPB);
```

Building the Graph, Cont'd

```cpp
// Node for apply jelly
continue_node<continue_msg> applyJelly(g,
   [](const continue_msg&) {
      std::cout << "Applying jelly...\n";
});

// Apply jelly is dependent on getting the bread
make_edge(getBread, applyJelly);
```

Building the Graph, Cont'd

```
// Node for combining the slices
continue_node<continue_msg> combineSlices(g,
   [](const continue_msg&) {
      std::cout << "Combining the slices...\n";
});

// Combining slices is dependent on BOTH apply nodes
make_edge(applyPB, combineSlices);
make_edge(applyJelly, combineSlices);
```

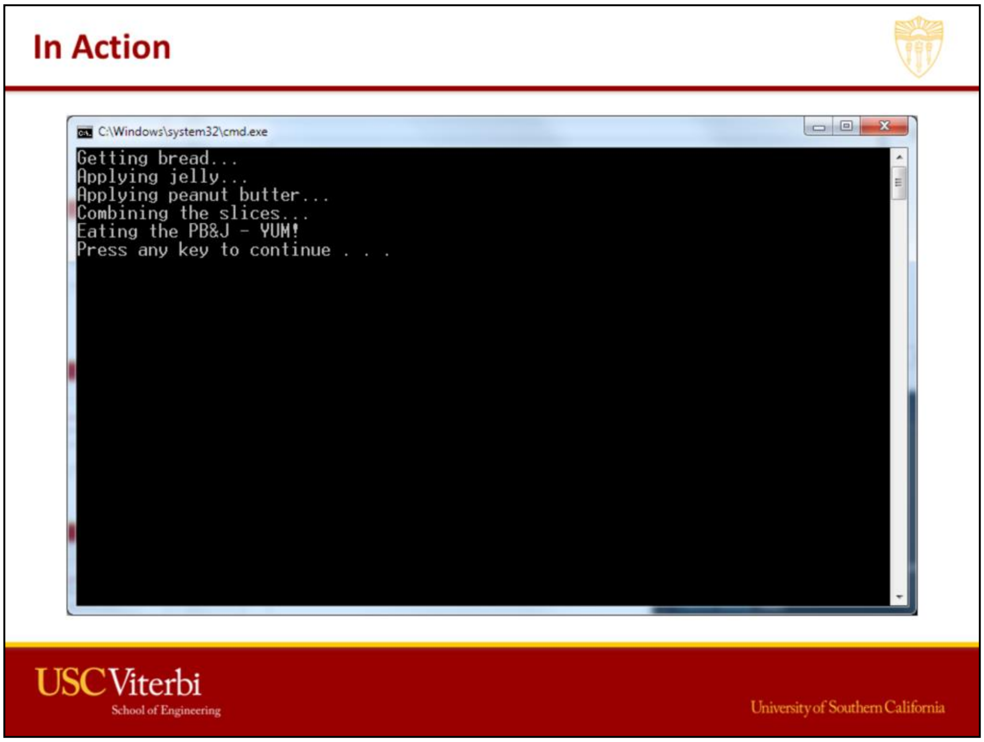
Building the Graph, Cont'd

```
// Node for eating the PB&J
continue_node<continue_msg> eatPBJ(g,
   [](const continue_msg&) {
      std::cout << "Eating the PB&J - YUM!\n";
});

// Eating is dependent on combining slices
make_edge(combineSlices, eatPBJ);
```

Executing the Graph

- Once the graph is ready to go, you can execute it by passing a continue_msg to the root node

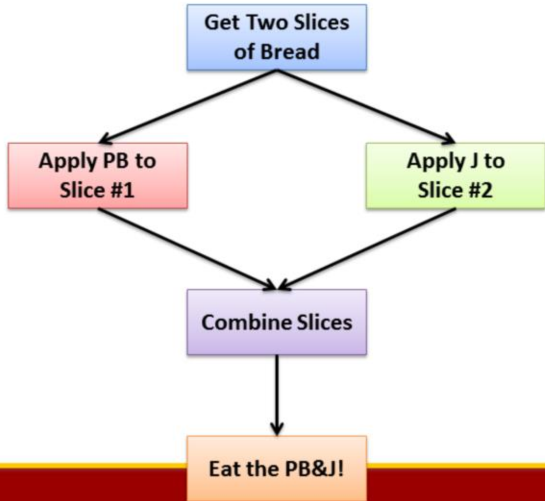
```
// Send a continue message to the first node
// to begin execution of the graph
getBread.try_put(continue_msg());
// Wait until all nodes finish
g.wait_for_all();
```



Notice how it executes the nodes while fulfilling all the dependencies

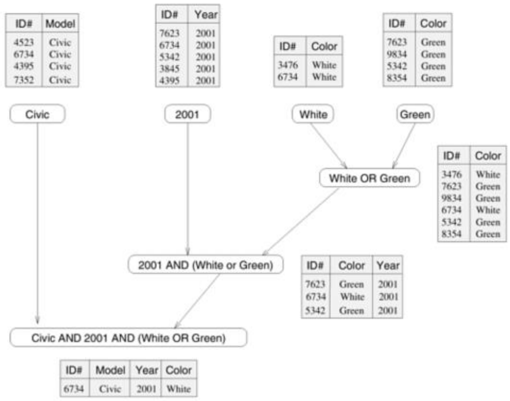
Of course…

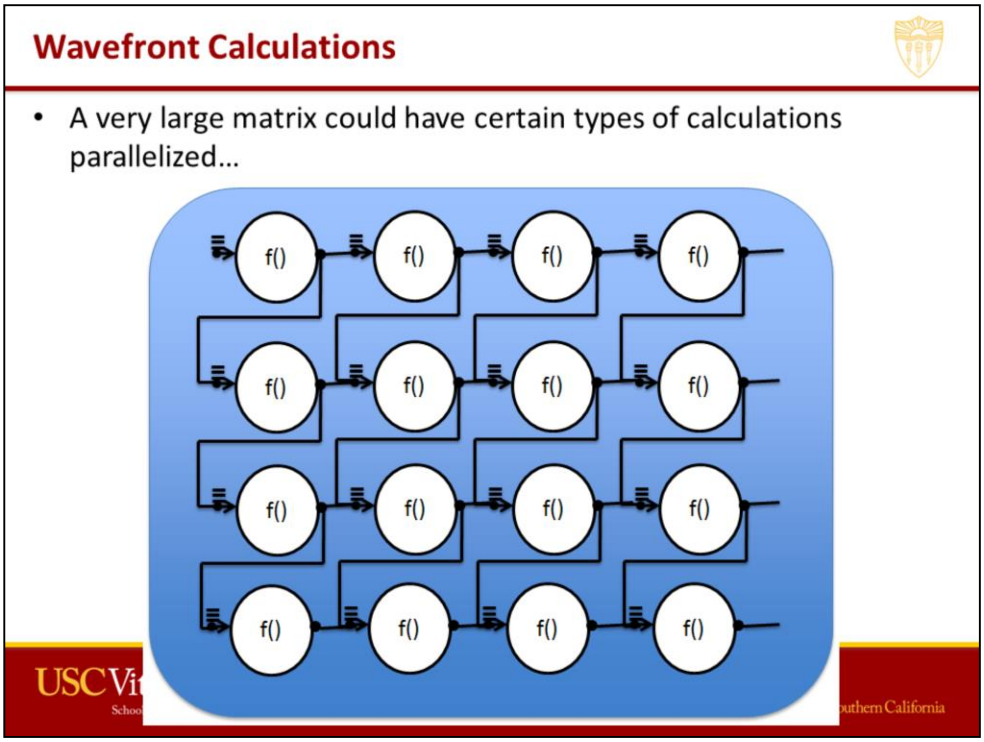
- This graph can't really be parallelized that much…



Database Query

- An SQL database query could be parallelized to some degree if broken down into its dependencies:





You could actually take this approach in PA3 if you wanted to, but it’s not a requirement since there are other challenges at hand

Other Types of Nodes

- TBB supports quite a few kinds of flow graph nodes, beyond just continue nodes: