Overview of Parallel Programming in C++

Pablo Halpern <pablo.g.halpern@intel.com>
Parallel Programming Languages Architect
Intel Corporation

CppCon, 8 September 2014

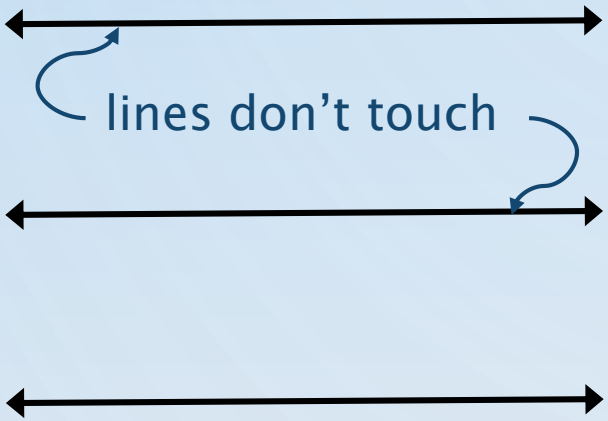
Questions to be answered

- What is parallel programming and why should I use it?
- How is parallelism different from concurrency?
- What are the basic tools for writing parallel programs in C++?
- What kinds of problems should I expect?

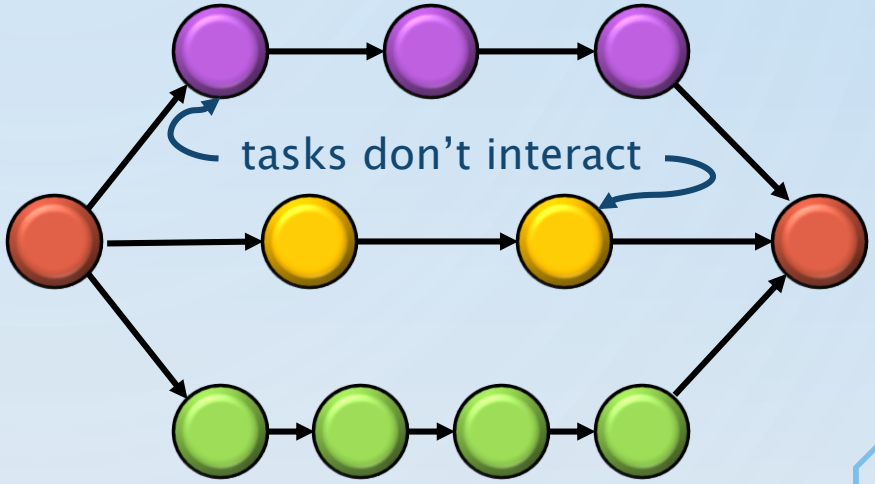
What and Why?

What is parallelism?

Parallel lines in geometry:



Parallel tasks in programming:

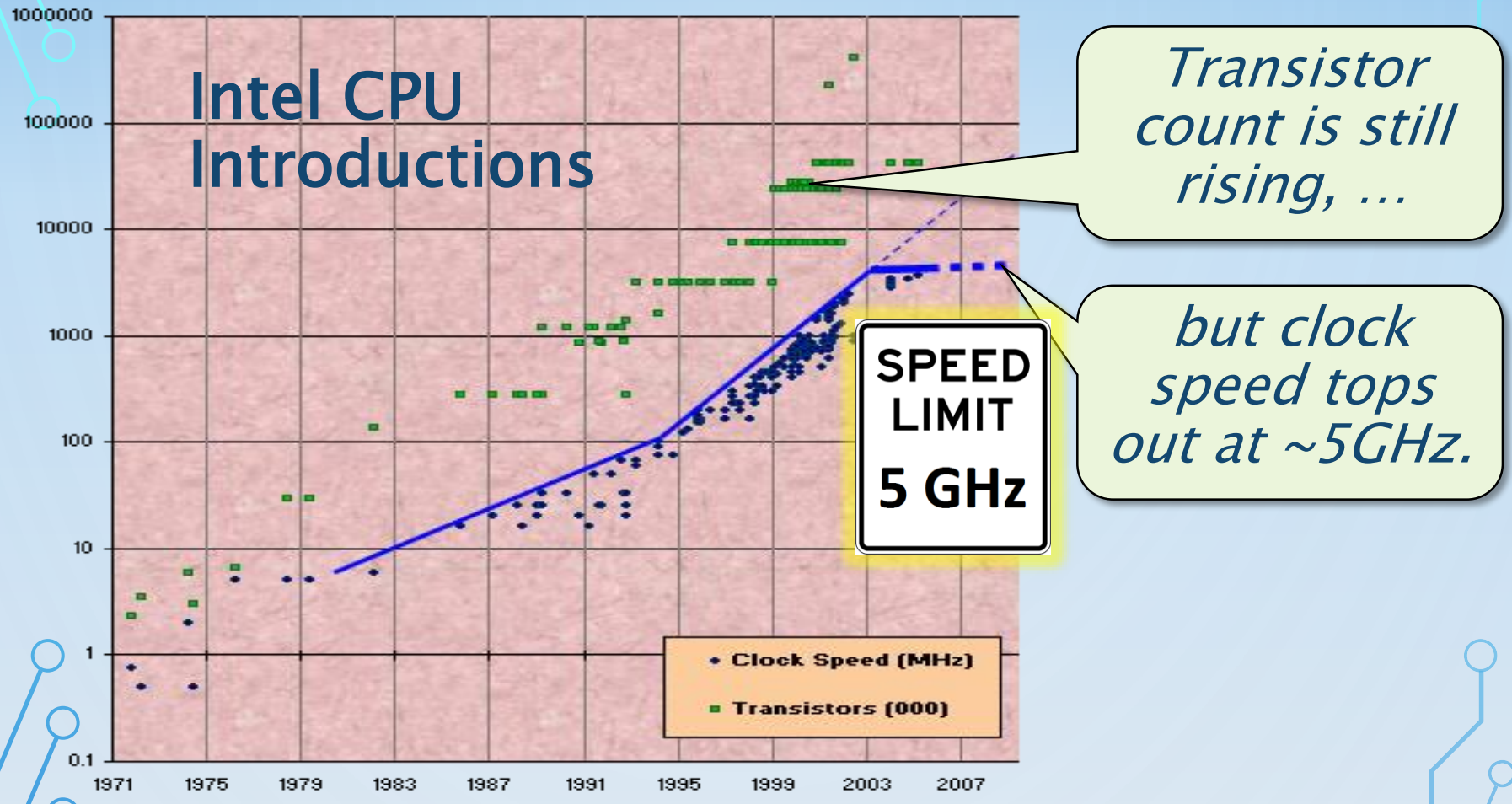


Why go parallel?

- Parallel programming is needed to efficiently exploit today's multicore hardware:
  - Increase throughput
  - Reduce latency
  - Reduce power consumption

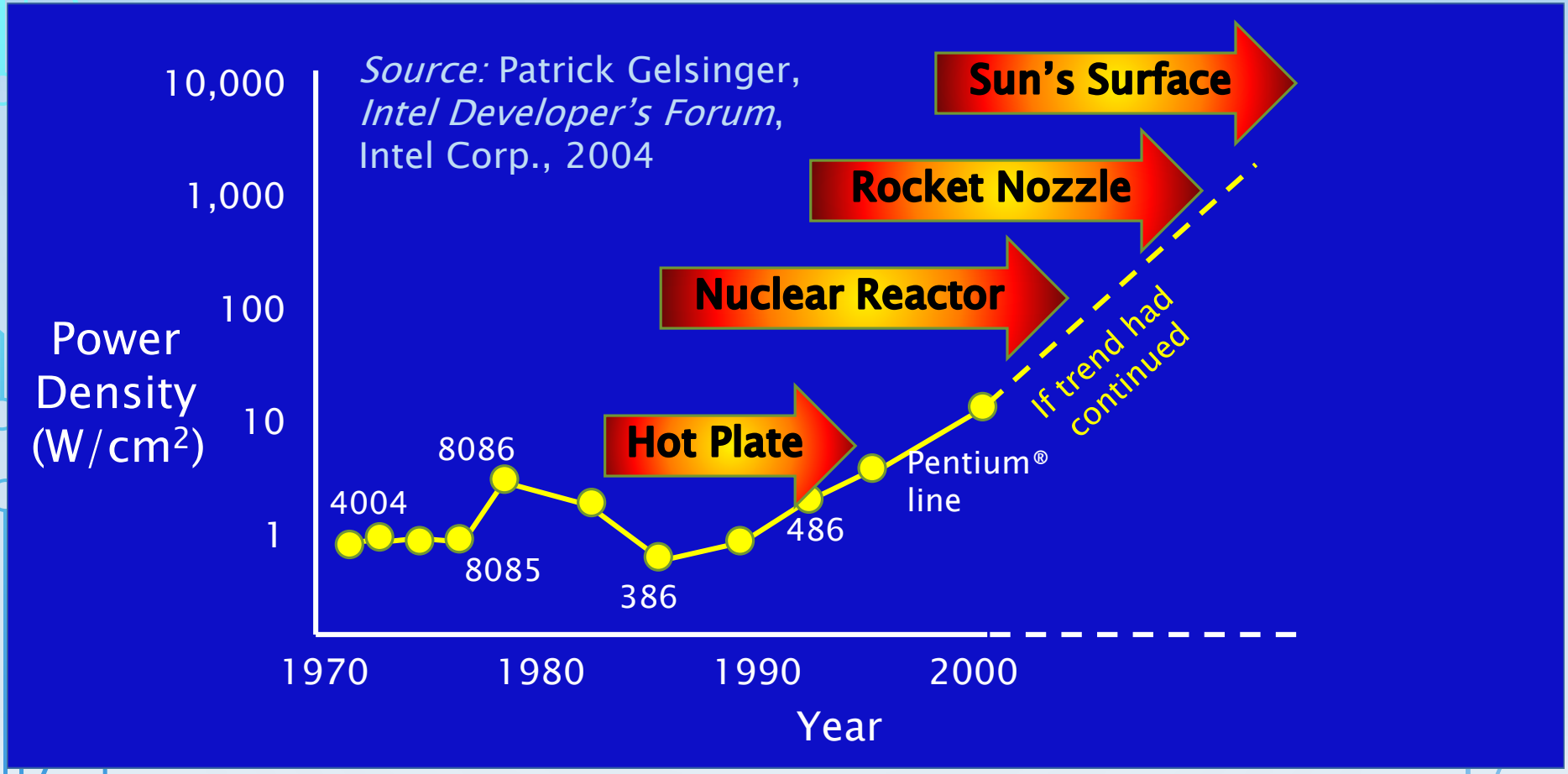
- But why did it become necessary?

Moore's Law

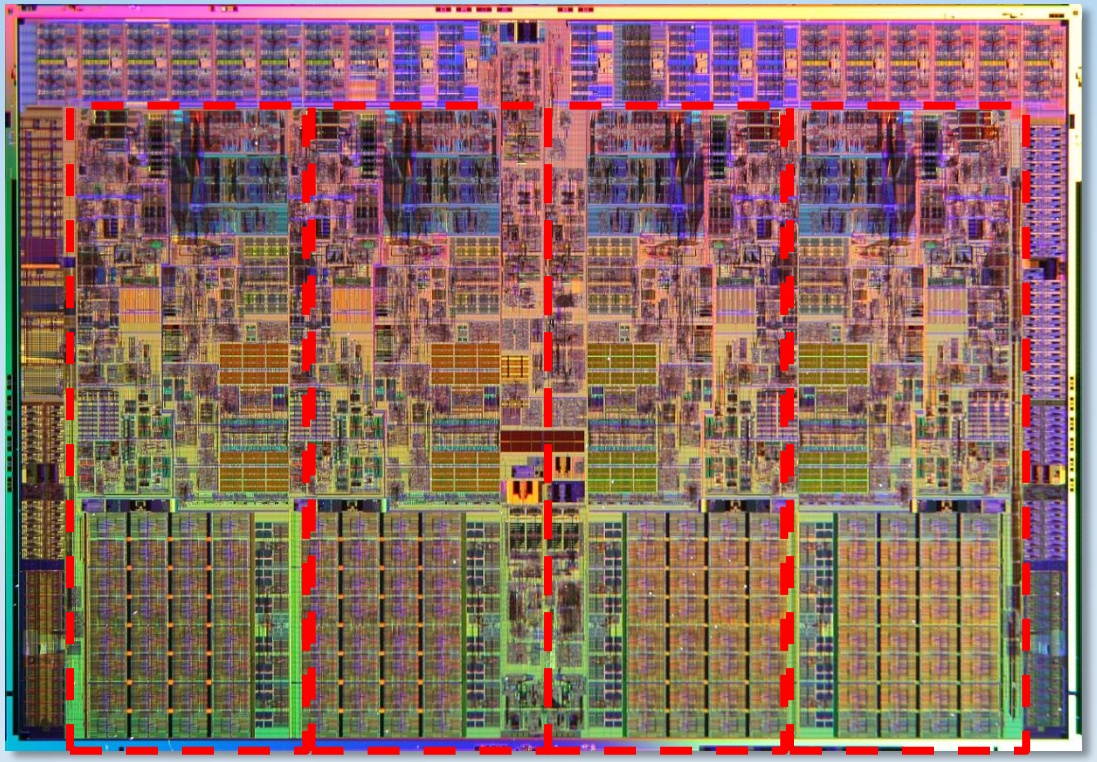


Source: Herb Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, 30(3), March 2005.

The single-core power/heat wall



Vendor solution: Multicore



Intel Core i7 processor

- 2 cores running at 2.5 GHz use less power and generate less heat than 1 Core at 5 GHz for the same GFLOPS.
- 4 cores are even better.

Concurrency and Parallelism

Concurrency and parallelism: They're not the same thing!

CONCURRENCY

- **Why:** express component interactions for effective *program structure*
- **How:** interacting *threads* that can wait on events or each other

PARALLELISM

- **Why:** exploit *hardware* efficiently to scale *performance*
- **How:** independent *tasks* that can run simultaneously

A program can have both

Sports analogy

Photo credit JJ Harrison (CC) BY-SA 3.0

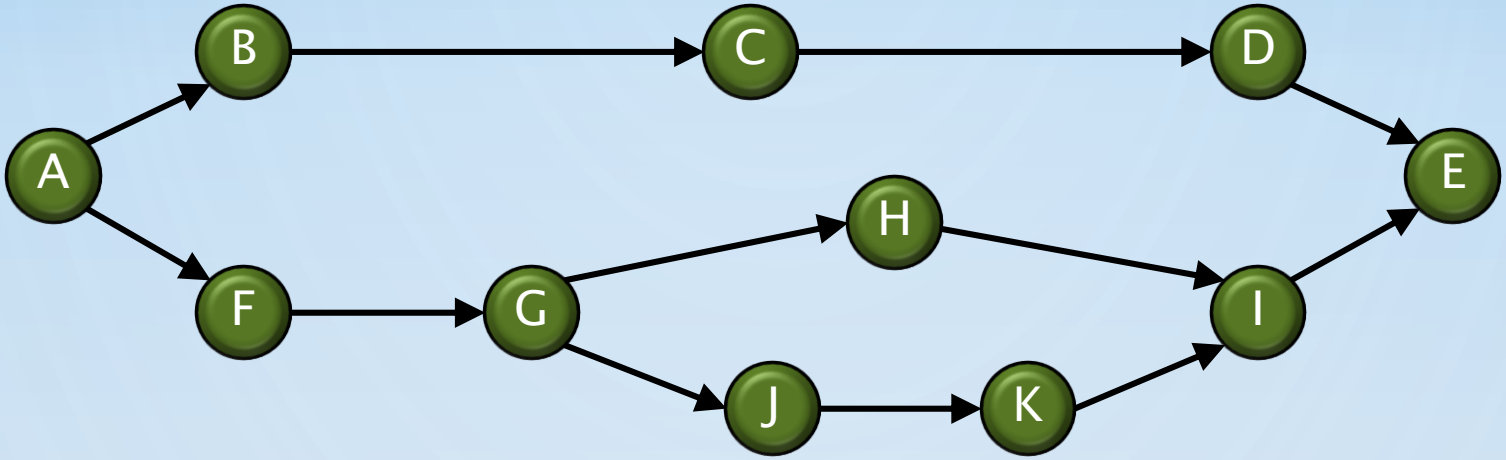
Photo credit André Zehetbauer (CC) BY-SA 2.0

Concurrency

Parallelism

Basic concepts and vocabulary

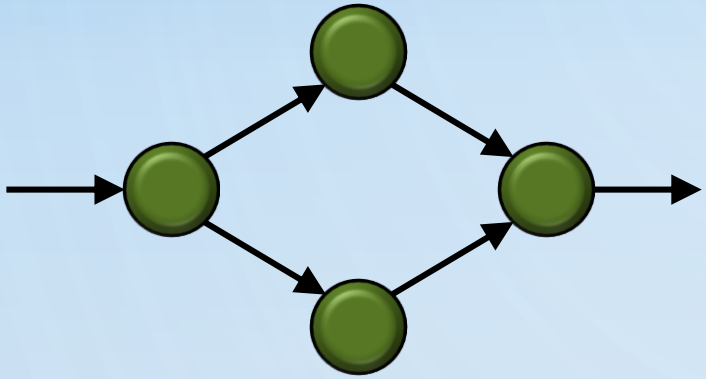
Parallelism is a graph-theoretical property of the algorithm



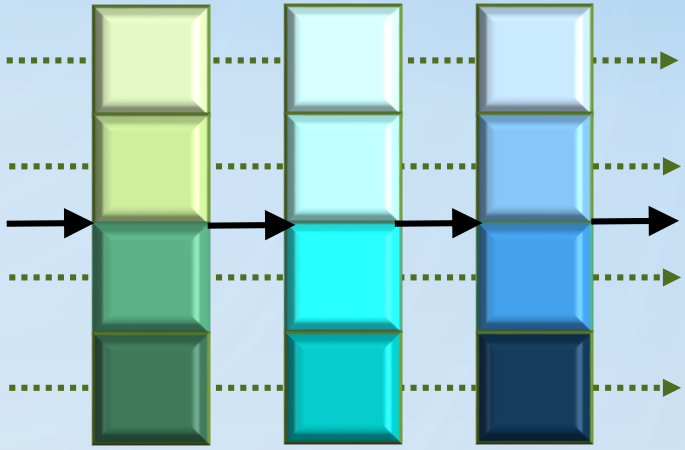
(Dependencies are opposite control flow, e.g. C depends on B)

- A $\prec$ B and A $\prec$ F (A *precedes* B and F)
- B $\parallel$ F (B is *in parallel with* F)
- K $\succ$ G (K *succeeds* G) and
- K $\parallel$ H, K $\parallel$ B and K $\parallel$ C, etc.

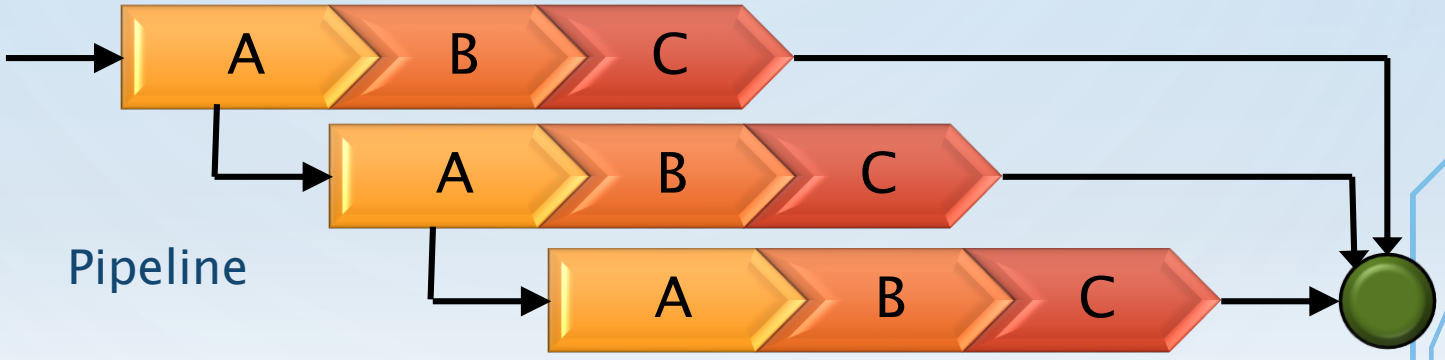
Types of parallelism



Fork-Join



Vector/SIMD



Pipeline

A modest example

The world's worst Fibonacci algorithm

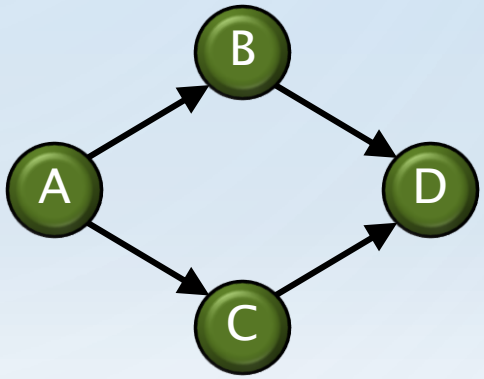
```
   int fib(int n)
   {
A      if (n < 2) return n;

B      int x = fib(n - 1);
C      int y = fib(n - 2);

D      return x + y;
   }
```

Dependency-graph analysis:

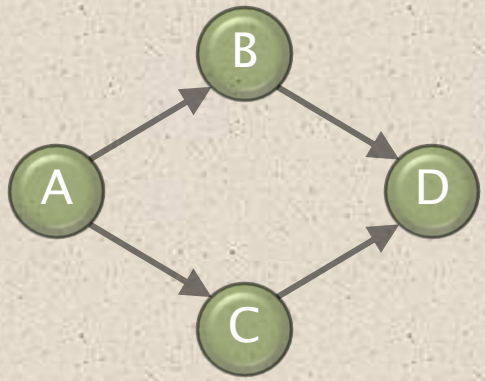
- A $\prec$ B and A $\prec$ C
- B $\parallel$ C
- B $\prec$ D and C $\prec$ D



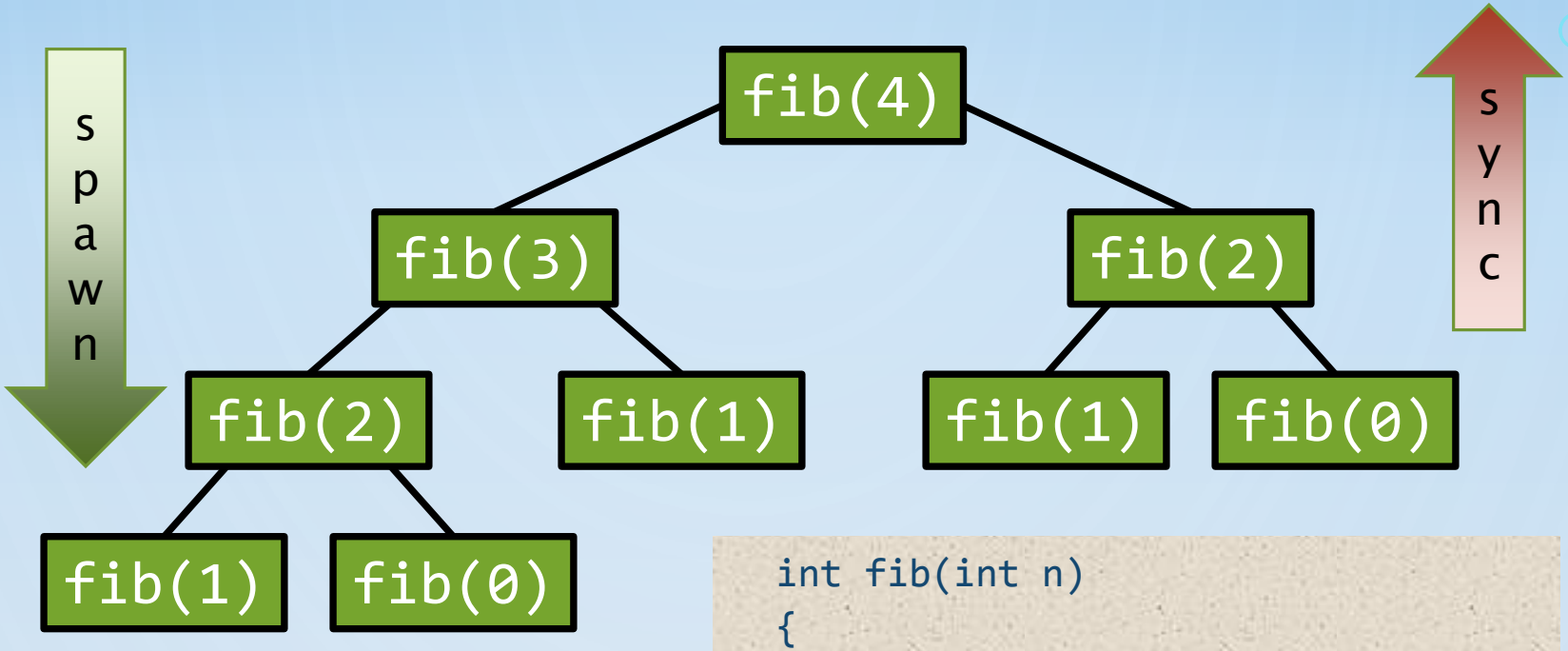
Parallelizing fib using Cilk™ Plus

```
   int fib(int n)
   {
A      if (n < 2) return n;

B      int x = cilk_spawn fib(n - 1);
C      int y = fib(n - 2);
       cilk_sync;
D      return x + y;
   }
```



Fibonacci Execution




```
int fib(int n)
{
    if (n < 2) return n;

    int x = cilk_spawn fib(n - 1);
    int y = fib(n - 2);
    cilk_sync;
    return x + y;
}
```

A more realistic example: Quicksort

```
template <typename Iter, typename Cmp>
void par_qsort(Iter begin, Iter end, Cmp comp)
{
    typedef typename std::iterator_traits<Iter>::value_type T;
    if (begin != end) {
        Iter pivot = end - 1; // For simplicity. Should be random.
        Iter middle = std::partition(begin, pivot,
            [=](const T& v){ return comp(v, *pivot); });
        using std::swap;
        swap(*pivot, *middle);      // move pivot to middle
        cilk_spawn par_qsort(begin, middle, comp);
        par_qsort(middle+1, end, comp); // exclude pivot
    }
} // implicit sync at end of function
```

Languages and libraries for parallel programming in C++

Parallelism Libraries: TBB and PPL

Fork-join parallelism

```
tbb::task_group tg;
tg.run([=]{ par_qsort(begin, middle, comp); });
tg.run([=]{ par_qsort(middle+1, end, comp); });
tg.wait();
```

```
tbb::parallel_for(0, n, [&](int i){
    f(i);
});
```

Pipeline parallelism

```
tbb::parallel_pipeline(16,
  make_filter<void, string>(filter::serial, gettoken) &
  make_filter<string, rec>(filter::parallel, lookup) &
  make_filter<rec, void>(filter::parallel, process));
```

Graph parallelism

```
tbb::graph g;
… /* Add nodes */
g.wait_for_all();
```

TBB Only

Parallelism pragmas: OpenMP

Fork–join parallelism

```
#pragma omp task
    par_qsort(begin, middle, comp);
#pragma omp task
    par_qsort(middle+1, end, comp);
#pragma omp taskwait
```

```
#pragma omp parallel for
for (int i = 0; i < n; ++i)
    f(i);
```

Vector parallelism

```
#pragma omp simd
for (int i = 0; i < n; ++i)
    f(i); // f() could be simd-enabled
```

Parallel language extensions: Cilk™ Plus

Fork–join parallelism

```
cilk_spawn par_qsort(begin, middle, comp);
par_qsort(middle+1, end, comp);
cilk_sync;
```

```
cilk_for (int i = 0; i < n; ++i)
    f(i);
```

Vector parallelism

```
#pragma simd
for (int i = 0; i < n; ++i)
    f(i);  // f() could be simd-enabled
```

```
extern float a[n], b[n];
a[:] += g(b[:]);  // g() could be simd-enabled
```

- Pipeline parallelism constructs are available as experimental software on the cilkplus.org web site.
- Cilk Plus supports *hyperobjects*, a unique feature to reduce data contention (especially races).

Future C++ standard library for parallelism

Fork–join parallelism

```
parallel::task_region([&](auto tr_handle)
{
    tr_handle.run([=]{ qsort(begin, middle, comp); });
    par_qsort(middle+1, end, comp);
}
```

```
parallel::for_each(parallel::par,
                   int_iter(0), int_iter(n),
                   [&](auto it){ f(*it); });
```

Vector parallelism

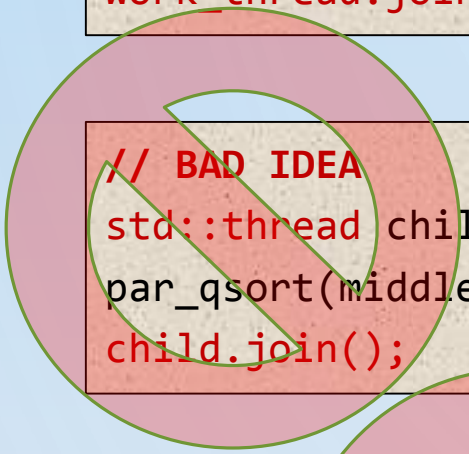
```
parallel::for_each(parallel::parvec,
                   int_iter(0), int_iter(n),
                   [&](auto it){ f(*it); });
```

```
for simd (int i = 0; i < n; ++i)
    f(i);  // f() could be simd-enabled
```

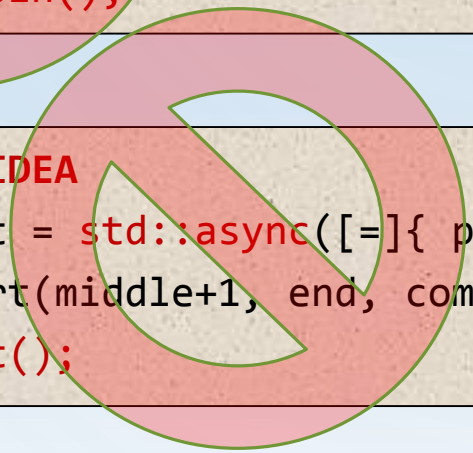
A draft Technical Specification (TS) also includes a parallel versions of STL algorithms.

C++ supports concurrency, too, but don't confuse it with parallelism!

```
// GOOD IDEA
std::thread work_thread(computeFunc);
event_loop();
work_thread.join();
```



```
// BAD IDEA
std::thread child([=]{ par_qsort(begin, middle, comp); });
par_qsort(middle+1, end, comp);
child.join();
```



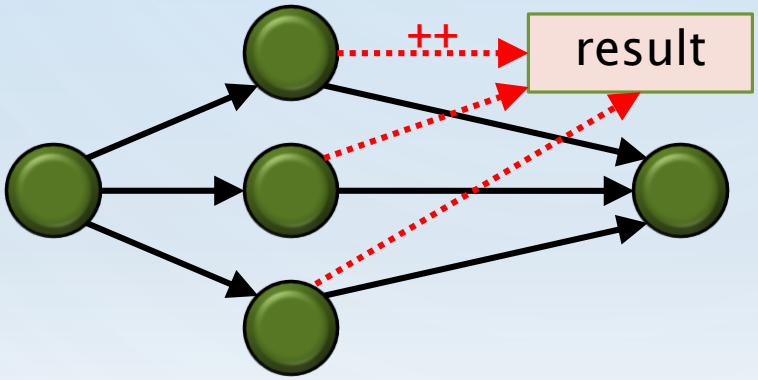
```
// BAD IDEA
auto fut = std::async([=]{ par_qsort(begin, middle, comp); });
par_qsort(middle+1, end, comp);
fut.wait();
```

Problems and Challenges

Data Races

```
template <class RandomIterator, class T>
size_t parallel_count(RandomIterator first, RandomIterator last,
                      const T& value) {
    size_t result(0);
    cilk_for (auto i = first; i != last; ++i)
        if (*i == value)
            ++result;
    return result;
}
```

Race!



Mitigating data races: Mutexes and atomics

```
std::mutex myMutex;
size_t result(0);
cilk_for (auto i = first;
          i != last; ++i)
    if (*i == value) {
        myMutex.lock();
        ++result;
        myMutex.unlock();
    }
```

Mutexes

```
std::atomic<size_t> result(0);
cilk_for (auto i = first;
          i != last; ++i)
    if (*i == value)
        ++result;
```

Atomics

Contention and overhead!

Mitigating data races: Reduction operations

```
cilk::reducer<cilk::op_add<size_t>> result(0);
cilk_for (auto i = first; i != last; ++i)
    if (*i == value)
        ++*result;
return result.get_value();
```

Cilk Plus reducer

```
size_t result(0);
#pragma omp parallel for reduction(+:result)
for (size_t i = 0; i != last - first; ++i)
    if (first[i] == value)
        ++result;
return result;
```

OpenMP reduction clause

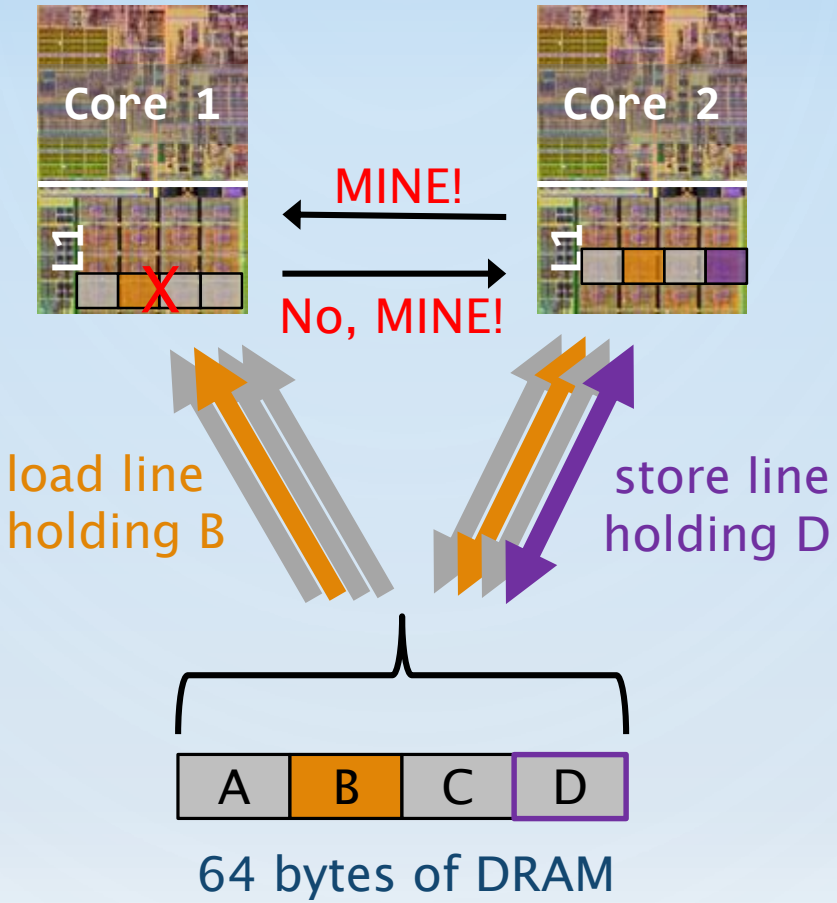
```
return tbb::parallel_reduce(...,
   if (*i == value) … ); // details elided
```

TBB reduce algorithm

Avoiding data races: Divide into disjoint data sets

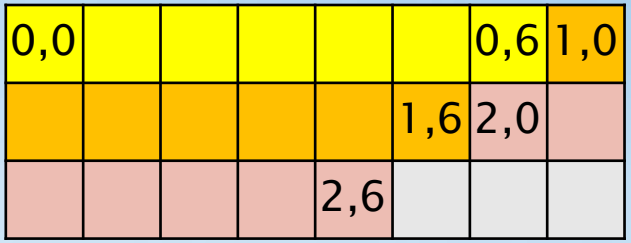
```
template <class RandomIterator, class T>
size_t parallel_count(RandomIterator first, RandomIterator last,
                      const T& value) {
    size_t result(0);
    if (last - first < 32) {
        for (auto i = first; i != last; ++i) // serial loop
            if (*i == value) ++result;
    } else {
        RandomIterator mid = first + (last - first) / 2;
        size_t a = cilk_spawn parallel_count(first, mid, value);
        size_t b = parallel_count(mid, last, value);
        cilk_sync;
        result = a + b;
    }
    return result;
}
```

Performance problem: False sharing



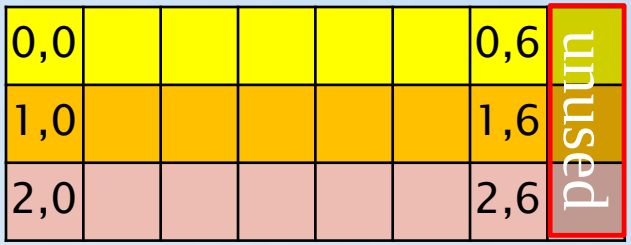
Avoiding false sharing

```
constexpr size_t M = 10000, N = 7;
double my_data[M][N];
…
cilk_for (size_t i = 0; i < M; ++i)
    modify_row(my_data[i]);
```



unaligned rows

```
constexpr size_t M = 10000, N = 7;
constexpr size_t cache_line = 64;
struct row {
  alignas(cache_line) double m[N];
};
row my_data[M];
```

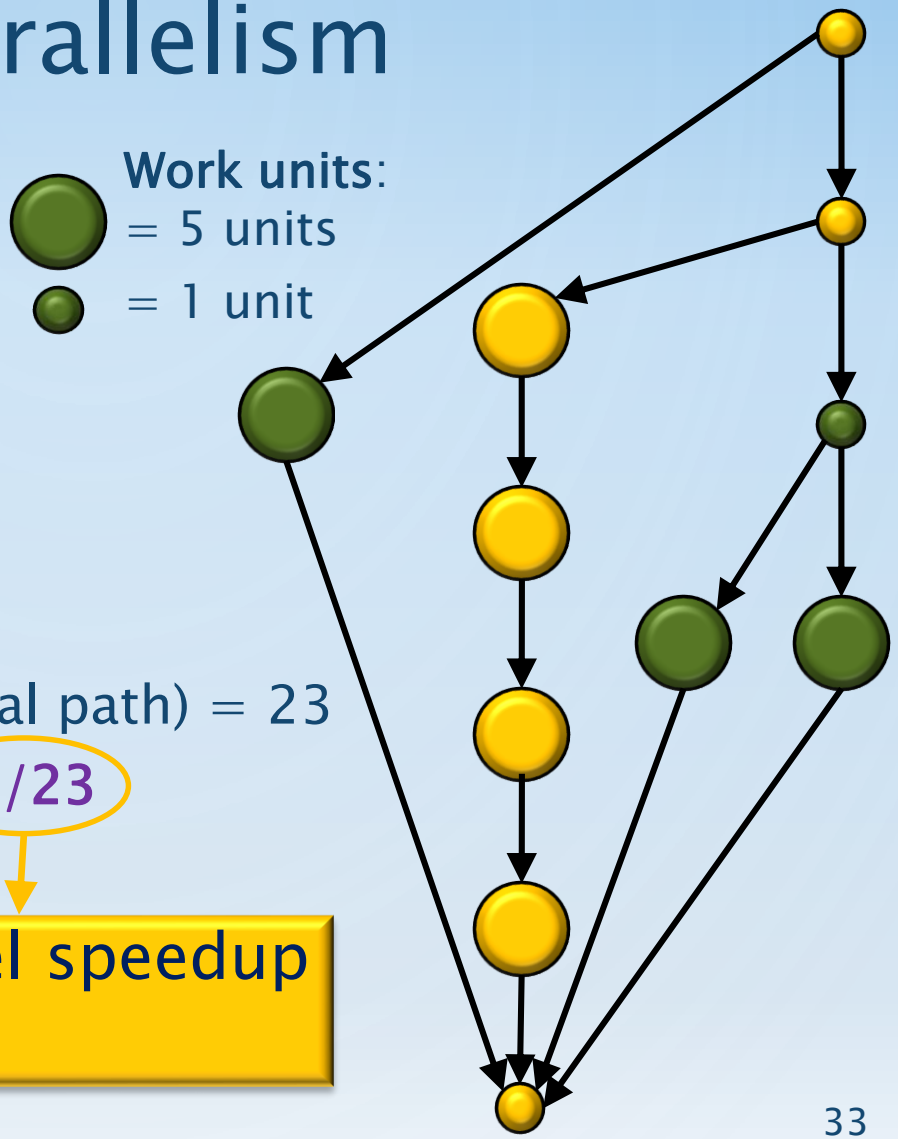


cache-aligned rows

```
constexpr size_t M = 10000, N = 7, cache_line = 64;
constexpr size_t N2 = ((N*sizeof(double) + cache_line-1) &
                       ~(cache_line-1)) / sizeof(double);
alignas(cache_line) double my_data[M][N2];
```

Performance bug: Insufficient parallelism

```
cilk_spawn short_func();
cilk_spawn long_func();
cilk_spawn short_func();
short_func();
cilk_sync;
```



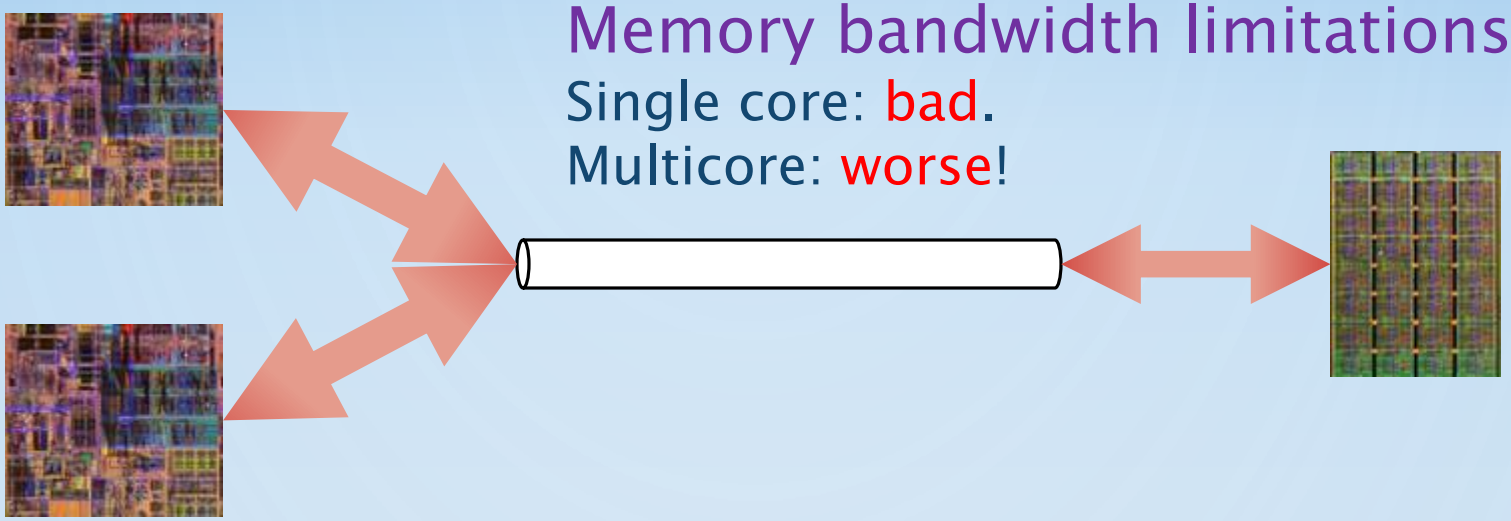
W = Total work = 39

S = Span (work on the critical path) = 23

P = Parallelism= W / S = 39/23

Maximum parallel speedup < 2

Serial challenges magnified



Memory bandwidth limitations:

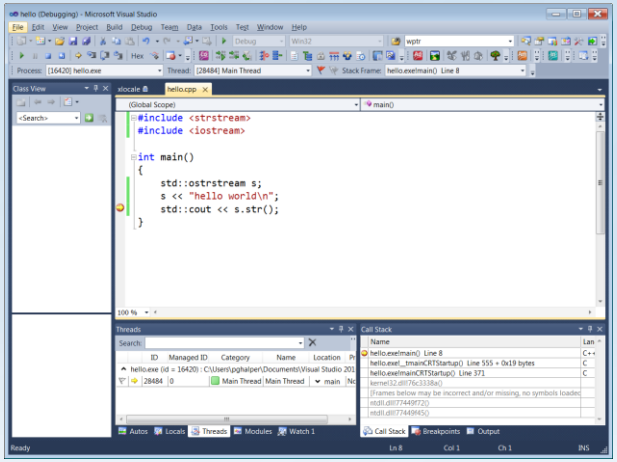
Single core: bad.

Multicore: worse!

Debugging:

Single thread: hard.

Multithread: harder!



Next steps

- Attend other CppCon sessions on parallelism, including my session on decomposing a problem for parallelism.
- Obtain a parallel compiler or framework and work through some tutorials.
- Get tools to help:
  - Race detector (Cilkscreen, Intel® Inspector XE, Valgrind)
  - Parallel performance analyzer (Cilkview, Cilkprof, Intel® VTune Amplifier XE)

Resources

- Intel® Cilk™ Plus (including downloads for Cilkscreen and Cilkview): cilkplus.org
- Intel® Threading Building Blocks (Intel® TBB): www.threadingbuildingblocks.org
- OpenMP: openmp.org
- Intel® Parallel Studio XE (includes VTune™ Amplifier and Inspector XE: https://software.intel.com/en-us/intel-parallel-studio-xe

Thank You!