Expressing parallelism in C++

Felice Pantaleo
CERN Experimental Physics Department

felice@cern.ch

Real-time feedback

- click here
- Typos, confused explanations, bad examples
- This is very important to ensure the best teaching standards!

You will learn...

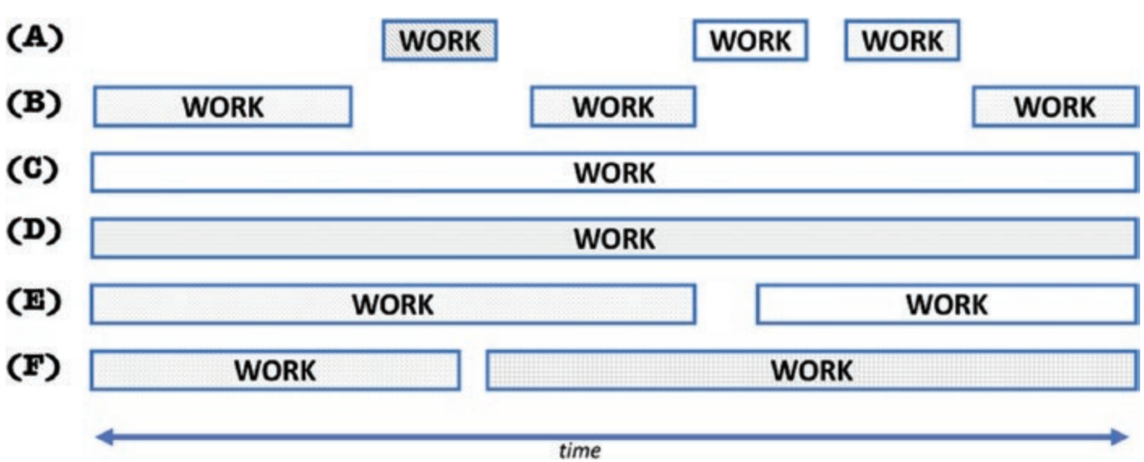
- Threads and Concurrency
- std::threads
- locks/mutual execution
- atomics
- Parallel algorithms
- Intel Threading Building Blocks
- Parallel execution with tbb
- Tasks parallelism

Threads

- A thread is an execution context, a set of register values
- Defines the instructions to be executed and their order
- A CPU core fetches this execution context and starts running the instructions: the thread is running
- When the CPU needs to execute another thread, it *switches the context* , *i.e.* saving the previous context and loading the new one
  - Context switching is expensive
  - Avoid threads jumping from a CPU core to another

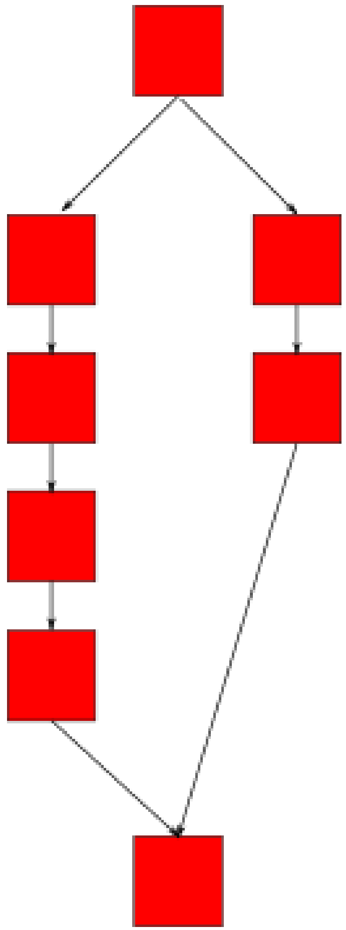
Threads enable concurrency

- Concurrency does not imply parallelism
- If your program contains independent parts, they are the perfect candidates for running concurrently
- Restaurant for dinner:
  - cooking food and preparing the tables are independent tasks and they can be performed by different workers to gain a speed-up
- A & B are concurrent wrt to each other and are also parallel wrt to C, D, E,F



Critical Path

- T = 1 is the time to compute a red box
- Serial Time = 8
- Span = 6
- Maximum speed-up = 8/6 ~ 1.33
- Speed up with 2 cores = 1.33
- Speed up with 100 cores = 1.33



std::threads – Hello World

```
#include <thread>
#include <iostream>
int main()
{


}
```

compile with

```
g++ std_threads.cpp -pthread -o std_threads
```

std::threads – Hello World

```
#include <thread>
#include <iostream>
int main()
{


}
```

Define a fuction that prints Hello world

```
void f(int i){
  std::cout << “Hello world from thread” << i <<
std::endl;
}
```

std::threads – Hello World

```
#include <thread>
#include <iostream>
int main()
{
   auto f = [](int i){
   std::cout << "hello world from thread " << i << std::endl;
  };
//Construct a thread which runs the function f
  std::thread t0(f,0);

//and then destroy it by joining it
  t0.join();
}
```

Congratulations!

- You have just written your first concurrent program
- Let's add some more threads and look at the output

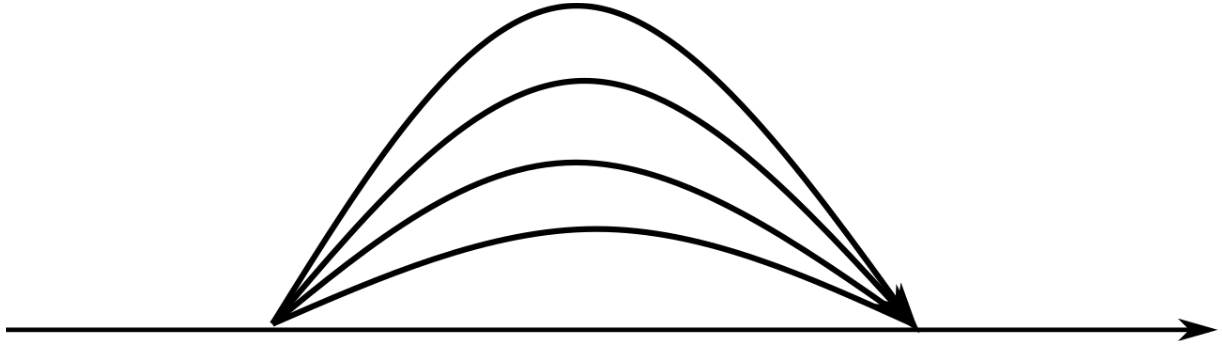
std::threads – Hello World

```
#include <thread>
#include <iostream>
int main()
{
   auto f = [](int i){
   std::cout << "hello world from thread " << i << std::endl;
  };
//Construct a thread which runs the function f
 std::thread t0(f,0);   std::thread t1(f,1);   std::thread t2(f,2);

//and then destroy it by joining it with the main thread
  t0.join(); t1.join(); t2.join();
}
```

Fork-join

- The construction of a thread is asynchronous, fork
- Threads execute independently
- join is the synchronization point with the main thread



Before we move on, measuring time

```
#include <chrono>

...

auto start = std::chrono::steady_clock::now();

  f(i);

auto stop = std::chrono::steady_clock::now();

std::chrono::duration<double> dur= stop - start;

std::cout << dur.count() << " seconds" << std::endl;
```

`f()` is the function that you want to measure

Be careful, asynchronous functions return immediately: remember to synchronize before stopping the timer.

Exercise 1

- You want to sum the elements of a vector in parallel using 4 threads
- Accumulate the sum in the variable `sum`
- Let's start by creating a thread
- Brainstorming time!

Data Race

A race condition occurs when multiple tasks read from and write to the same memory without proper synchronization.

- The "race" may finish correctly sometimes and therefore complete without errors, and at other times it may finish incorrectly.
- If a data race occurs, the behavior of the program is undefined.

std::mutex

- Avoiding that multiple threads access a shared variable
- Use it together with a scoped lock:

```
#include <mutex>
std::mutex myMutex;
...
{
  std::lock_guard<std::mutex> myLock(myMutex);
  //critical section begins here
  std::cout << "Only one thread at a time" << std::endl;
} // ends at the end of the scope of myLock
```

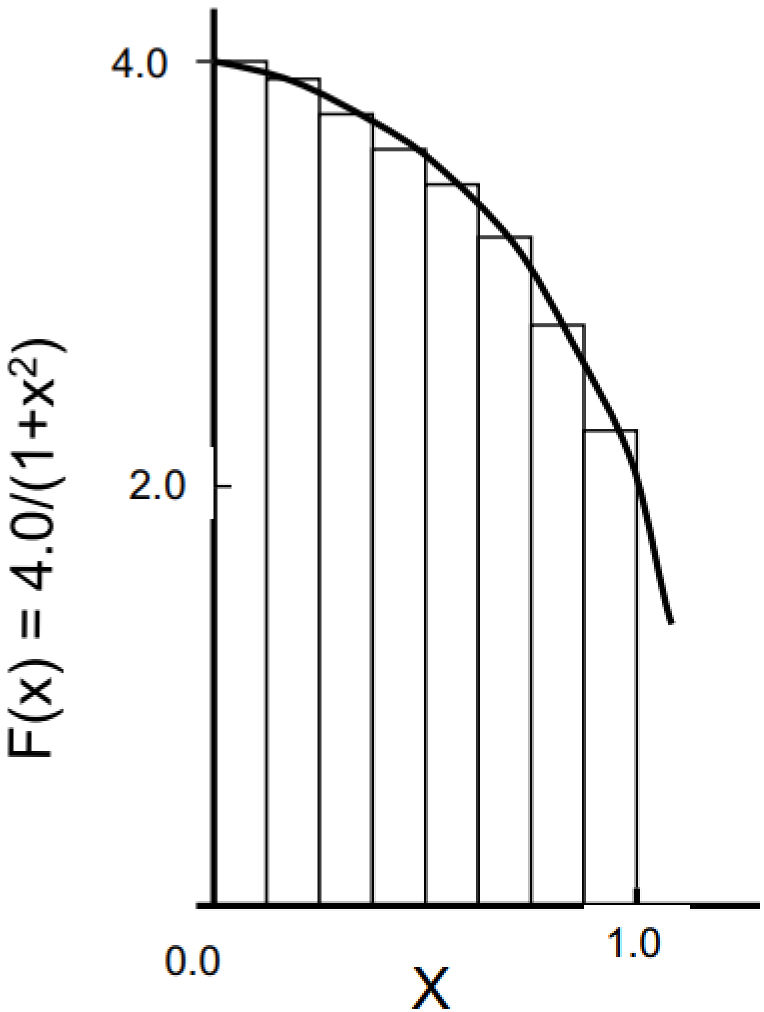
Some measurements

- Now you're ready to increase the number of threads!
- Time vs number of threads?
- Effect of privatization?

- Hint for creating multiple threads:

```
auto n = std::thread::hardware_concurrency();
std::vector<std::thread> v;
for (auto i = 0; i < n; ++i) {
     v.emplace_back(f,i);
}
for (auto& t : v) {
    t.join();
}
```

Exercise 2 - Numerical Integration



We know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

- The integral can be approximated as the s angles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Numerical integration

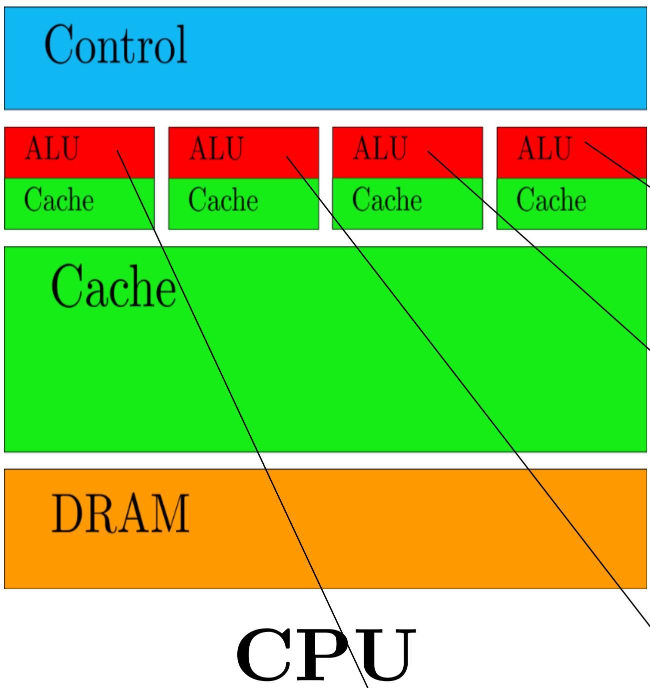
```
constexpr int num_steps = 1<<20;
double pi = 0.;
constexpr double step = 1.0/(double) num_steps;
double sum = 0.;

for (int i=0; i< num_steps; i++){
  auto x = (i+0.5)*step;
  sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;

std::cout << "result: " << std::setprecision (15) << pi << std::endl;
```

- Try to parallelize it
- Measure time vs number of threads, vs number of steps, play with parameters and check precision
- Try privatization
- What happens if one thread runs over more steps than the others? Why?

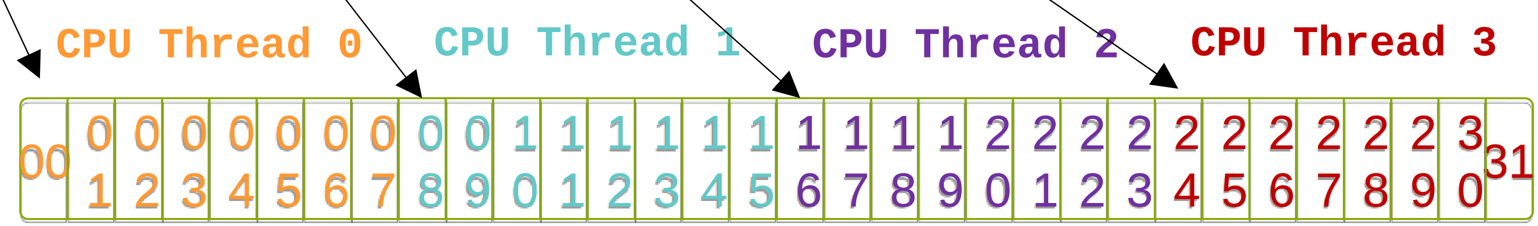
Memory access patterns: cached



Effective parallel programming requires that we have a sense of the importance of locality.

For optimal CPU cache utilization, the thread a should process element i and i+*1*

- stride=1

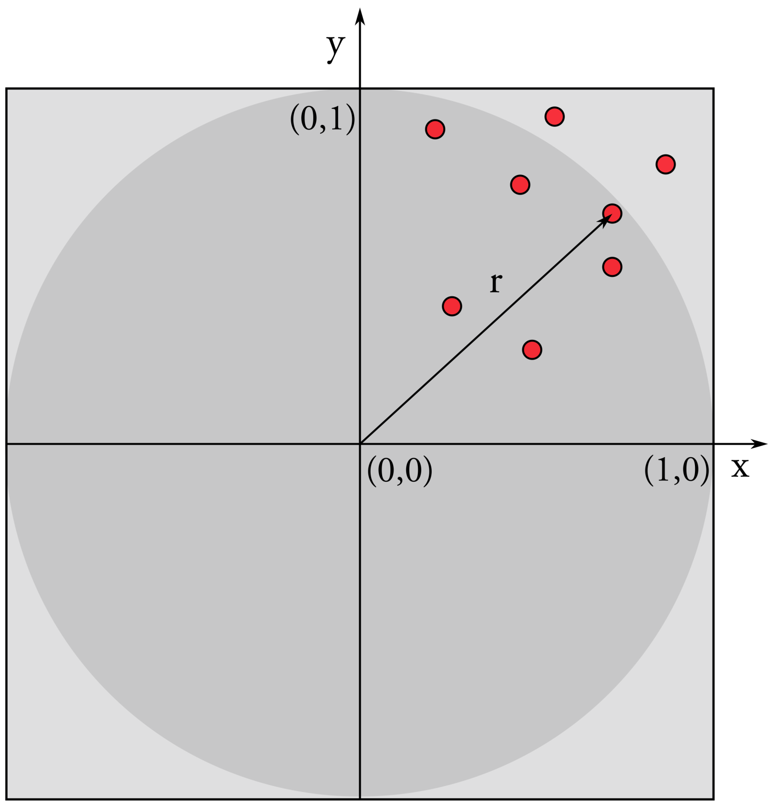


False Sharing

- Suppose that:
  - a cache line is 64bytes
  - two threads (x and y) run on processors that share their cache
  - we have two arrays int `A[500], B[500]`
  - the end of A and the beginning of B are in the same cache line
  - thread x modifies `A[499]`, and loads the corresponding cache-line in cache
  - thread y modifies `B[0]`
- The processor needs to flush the cache lines, reloading the cache for thread x and invalidating the cache for thread y
- Solution: align/padding to cache-line size

Exercise 3 - π with Monte Carlo

- The area of the circle is π
- The area of the square is 4
- Generate N random x and y between -1 and 1:
  - if r < 1: the point is inside the circle and increase N_{in}
  - The ratio between N_{in} and N converges to the ratio between the areas



std::atomic

- Atomic types:
  - encapsulate a value whose access is guaranteed to not cause data races
  - other threads will see the state of the system before the operation started or after it finished, but cannot see any intermediate state
  - can be used to synchronize memory accesses among different threads
  - at the low level, atomic operations are special hardware instructions
  - (hardware guarantees atomicity)
- The primary std::atomic template may be instantiated with any TriviallyCopyable type T
- Common architectures have atomic fetch-and-add instructions for integers

```
#include <atomic>
std::atomic<int> x = 0; int a = x.fetch_add(42);
```

- reads from a shared variable, adds 42 to it, and writes the result back: **all in one indivisible step**

Trivially Copyable

- Trivially copyable
- The primary `std::atomic` template may be instantiated with any TriviallyCopyable type T
  - Continuous chunk of memory
  - Copying the object means copying all bits (memcpy)
  - No virtual functions, noexcept constructor

```
std::atomic<int> i; // OK
std::atomic<double> x; // OK
struct S { long x; long y; };
std::atomic<S> s; // OK!
```

std::atomic<T>

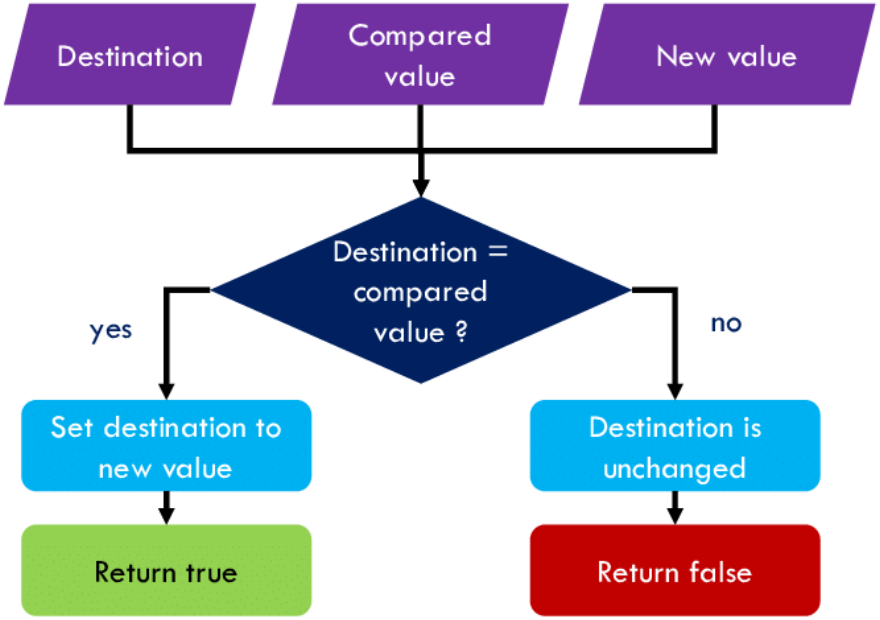
- read and write operations are always atomic
- `std::atomic<T>` provides operator overloads only for atomic operations (incorrect code does not compile)

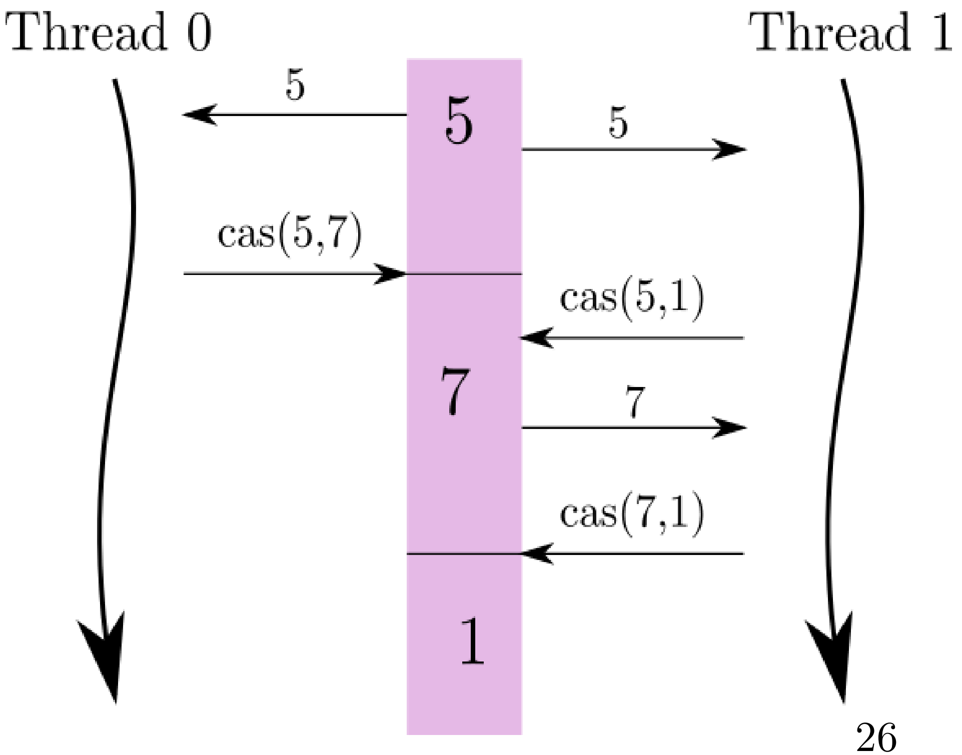
```
std::atomic<int> x{0}
++x;
x++;
x += 1;
x |= 2;
x *= 2; //this is not atomic and will not compile
int y = x * 2; // atomic read of x
x = y + 1; // atomic write of x
x = x + 1; // atomic read and then atomic write
x = x * 2; // atomic read and then atomic write
int z = x.exchange(y); // Atomically: z = x; x = y;
```

Compare-and-swap (CAS)

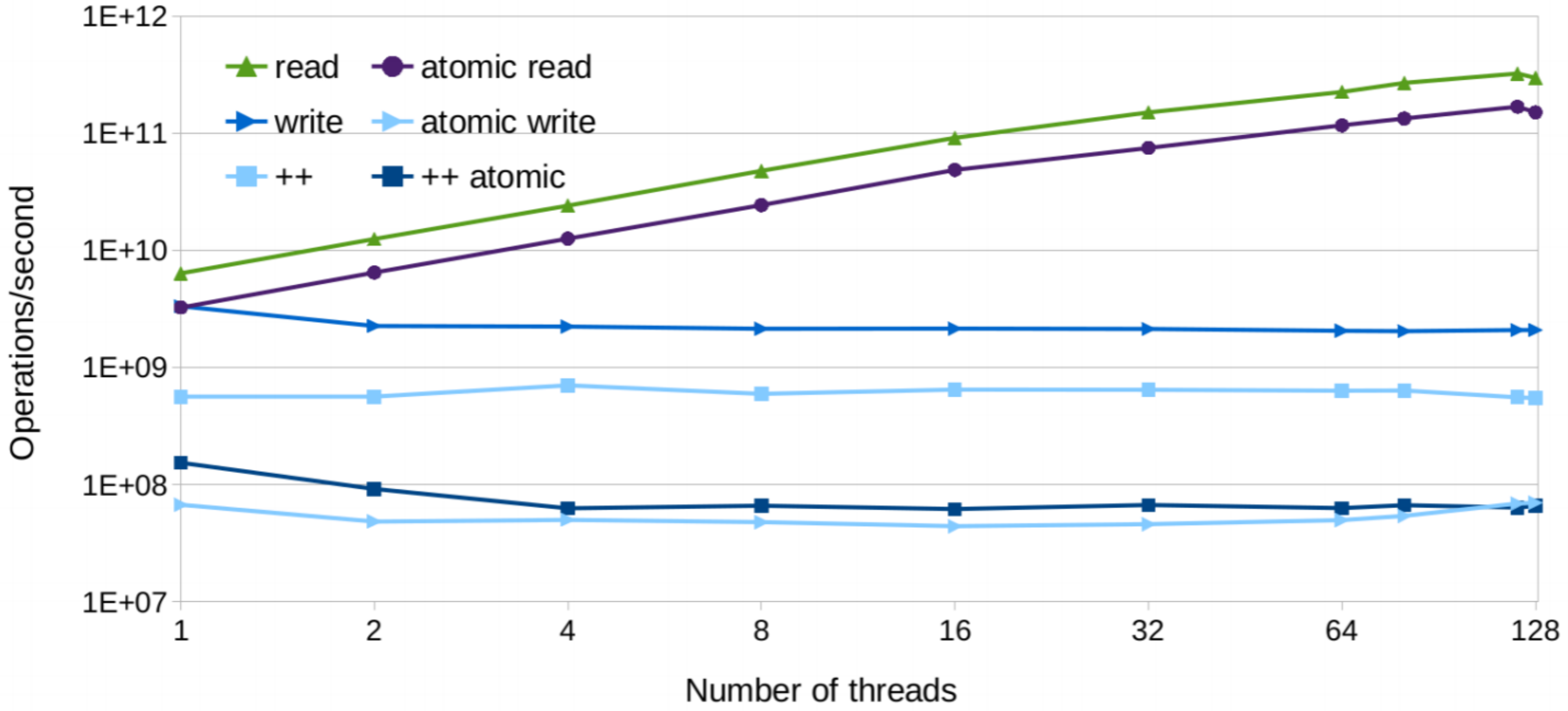
```
bool success = x.compare_exchange_weak(y, z);
```

```
uint32_t fetch_multiply(std::atomic<uint32_t>& shared, uint32_t multiplier)
{
    uint32_t oldValue = shared.load();
    while (!shared.compare_exchange_weak(oldValue, oldValue * multiplier)) { }
    return oldValue;
}
```



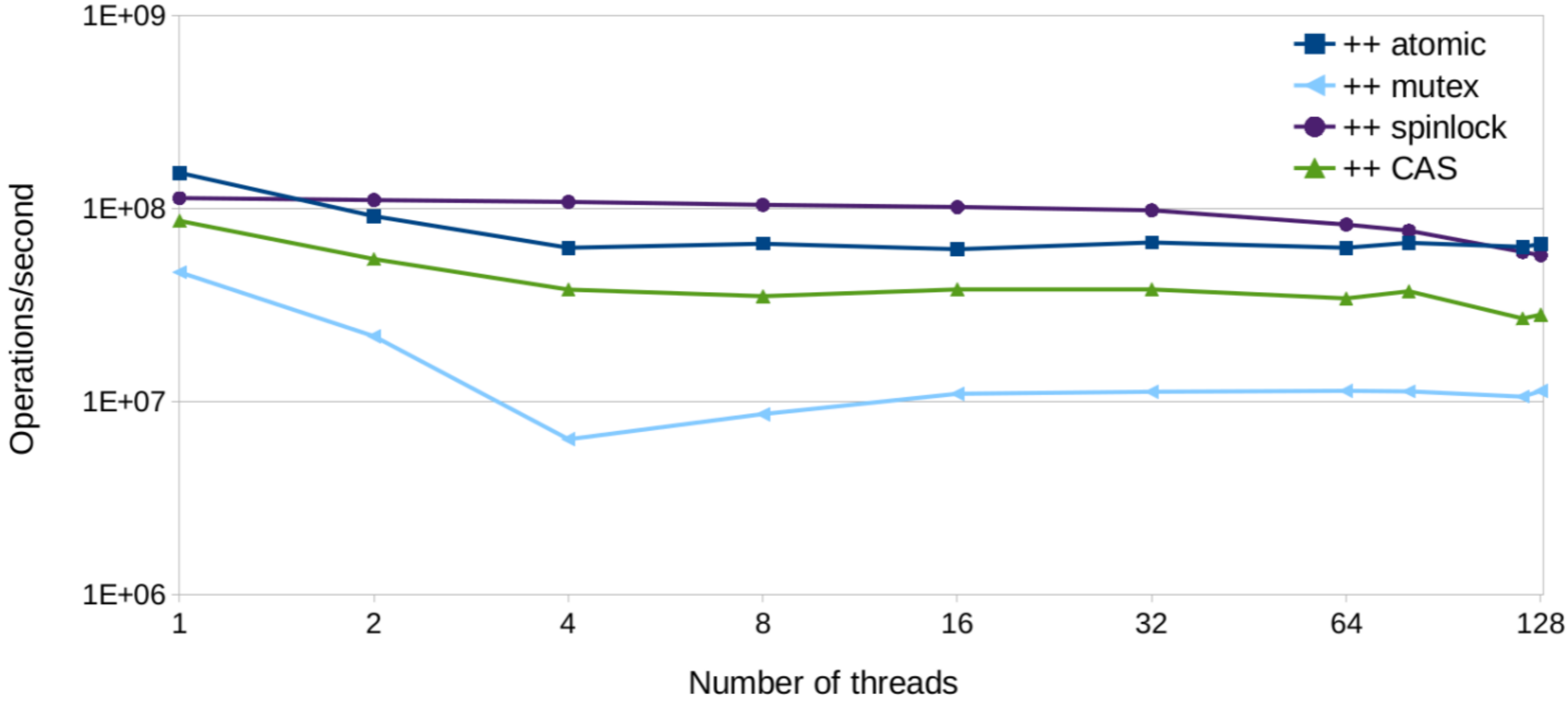


Are atomic operations slower than non-atomic?



Mentor A Siemens Business

Remember CAS?



Mentor
A Siemens Business

Expressing Parallelism with Intel Threading Building Blocks

Why TBB?

- Intel OneAPI Threading Building Blocks is a library which allows to express parallelism on CPUs in a C++ program
- Parallelizing for loops can be tedious with std::threads
- One wants to achieve *scalable* parallelism, easily
- To use the TBB library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner

Why TBB?

- Direct programming with threads forces you to do the work to efficiently map logical tasks onto threads
- TBB Runtime library maps tasks onto threads to maximize load balancing and squeezing performance out of the processor
  - Better portability
  - Easier programming
  - More understandable source code
  - Better performance and scalability

TBB Threads

```cpp
// Get the default number of threads
int num_threads = oneapi::tbb::info::default_concurrency();

// Run the default parallelism
oneapi::tbb::parallel_for(
        oneapi::tbb::blocked_range<size_t>(0, 20),
        [=](const oneapi::tbb::blocked_range<size_t> &r) {
          // Assert the maximum number of threads
          assert(num_threads == oneapi::tbb::this_task_arena::max_concurrency());
        });

// Create the default task_arena
oneapi::tbb::task_arena arena;
arena.execute([=] {
    oneapi::tbb::parallel_for(
        oneapi::tbb::blocked_range<size_t>(0, 20),
        [=](const oneapi::tbb::blocked_range<size_t> &r) {
          // Assert the maximum number of threads
          assert(num_threads ==
                 oneapi::tbb::this_task_arena::max_concurrency());
        });
});
```

Compile:

```
g++ hello_world.cpp -ltbb
```

Thread pool

A number of threads will be reused throughout your application to avoid the overhead of spawning them (or spawning too many)

```
// analogous to hardware_concurrency, number of hw threads:
int num_threads = oneapi::tbb::info::default_concurrency();

// or if you wish to force a number of threads:
auto t = 10; //running with 10 threads
oneapi::tbb::task_arena arena(t);

// And query an arena for the number of threads used:
auto max = oneapi::tbb::this_task_arena::max_concurrency();
// Limit the number of threads to two for all oneTBB parallel interfaces
oneapi::tbb::global_control global_limit(oneapi::tbb::global_control::max_allowed_parallelism, 2);
```

Parallelizing for loops with tbb

```
for(int i =0; i<N; ++i)   x[i]++;
```

becomes

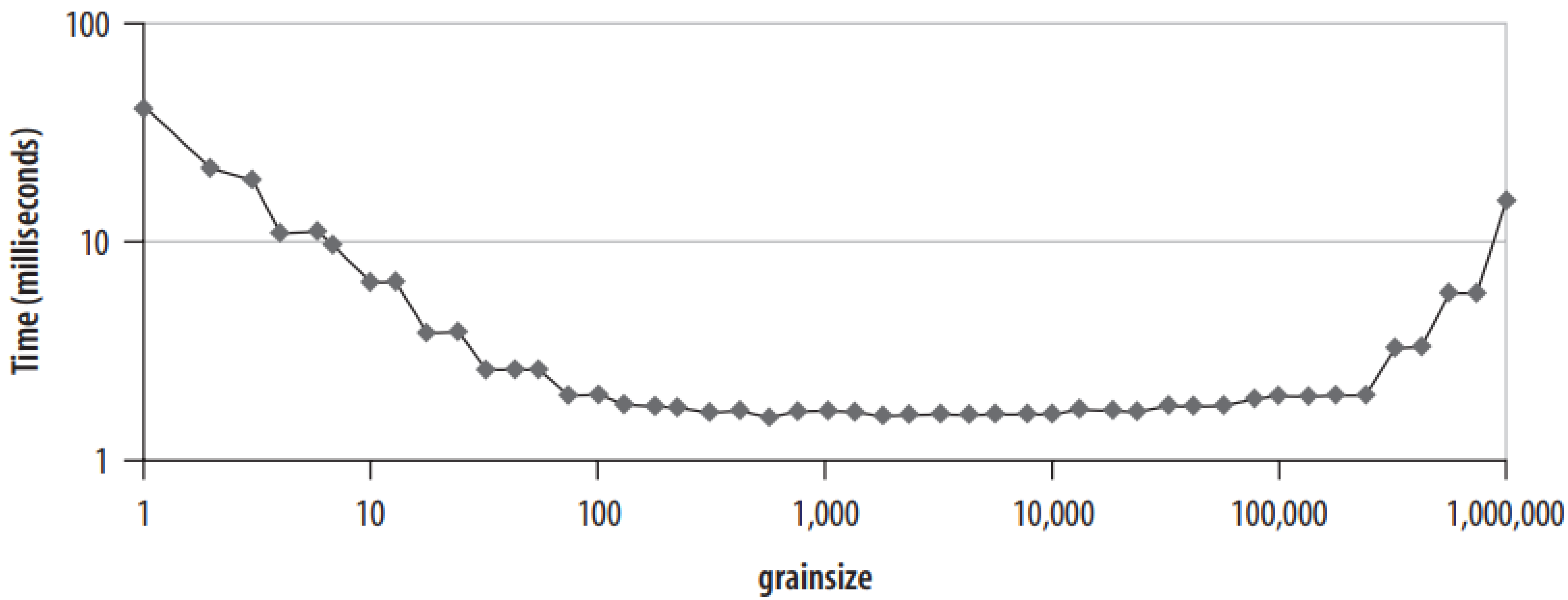
```
oneapi::tbb::parallel_for(
     oneapi::tbb::blocked_range<int>(0,N,<G>),
     [&](const oneapi::tbb::blocked_range<int>& range)
     {
         for(int i = range.begin(); i< range.end(); ++i)
         {
           x[i]++;
         }
     }, <partitioner>);
```

Scalability

- A loop needs to last for at least 1M clock cycles for parallel_for to become worth it
- If the performance of your application improves by increasing the number of cores, the application is said to *scale strongly*. There is usually a limit to the scaling.
- Usually, adding more cores than the limit does not only result in performance improvements, but performance falls.
  - Overhead in scheduling and synchronizing many small tasks starts dominating
- TBB uses the concept of *Grain Size* to keep data splitting to a reasonable level

Grain Size

- If GrainSize is 1000 and the loop iterates over 2000 elements, the scheduler can distribute the work at most to 2 processors
- With a GrainSize of 1, most of the time is spent



Automatic Partitioner

- The automatic partitioner is often more than enough to have good performance
- Heuristics that:
  - Limits overhead coming from small grain size
  - Creates opportunities for load balancing given by not choosing a grain size which is too large
- Sometimes controlling the grainSize can lead to performance improvements

Partitioners

- affinity_partitioner can improve performance when:
  - data in a loop fits in cache
  - the ratio between computations and memory accesses is low
- simple_partitioner enables the manual ninja mode
  - You need to specify manually the grain size G
  - The default is 1, in units of loop iterations per chunk
  - Rule of thumb: G iterations should take at least 100k clock cycles

Mutex Flavors

- Scalability
  - Not scalable if the waiting threads consume excessive processor cycles and memory bandwidth, reducing the speed of threads trying to do real work
- Fairness
  - Serves threads in the order they arrived (`queuing_mutex`)
- Yielding or Blocking
  - Yield: repeatedly poll, if no work allowed temporarily yield the processor
  - Block: yield the processor until the mutex permits progress

Mutex

- Header: `#include <oneapi/tbb/mutex.h>`
- Wrapper around OS calls:
  - Portable across all operating systems supported by TBB
  - Releases the lock if an exception is thrown from the protected region of code
- Usage:

```
oneapi::tbb::mutex myMutex;
...
{
  oneapi::tbb::scoped_lock myLock( myMutex );
  //critical section here
   …
}
```

- If the lock is lightly contended and the duration of the critical section is small, use `spin_mutex`
  - thread busy waits for lock to be released

```
oneapi::tbb::spin_mutex  myMutex;
...
{
oneapi::tbb::spin_mutex::scoped_lock myLock( myMutex );
//critical section here
…
}
```

Exercises 2 and 3 with tbb

- Try replacing `std::threads` with a `oneapi::tbb::parallel_for` in exercises 2 and 3
- Measure time to determine strong and weak scaling

Concurrent containers

- Concurrent containers allow concurrent thread-safe read-write access by multiple threads

```
oneapi::tbb::concurrent_vector<T>
oneapi::tbb::concurrent_queue<T>
oneapi::tbb::concurrent_hashmap<Key,T,HashCompare>
```

- For example:

```
#include <oneapi/tbb/concurrent_vector.h>
…
oneapi::tbb::concurrent_vector<int> myVector;
… // later in a parallel section
myVector.push_back(x);
```

Exercise 4 - Parallel Histogram

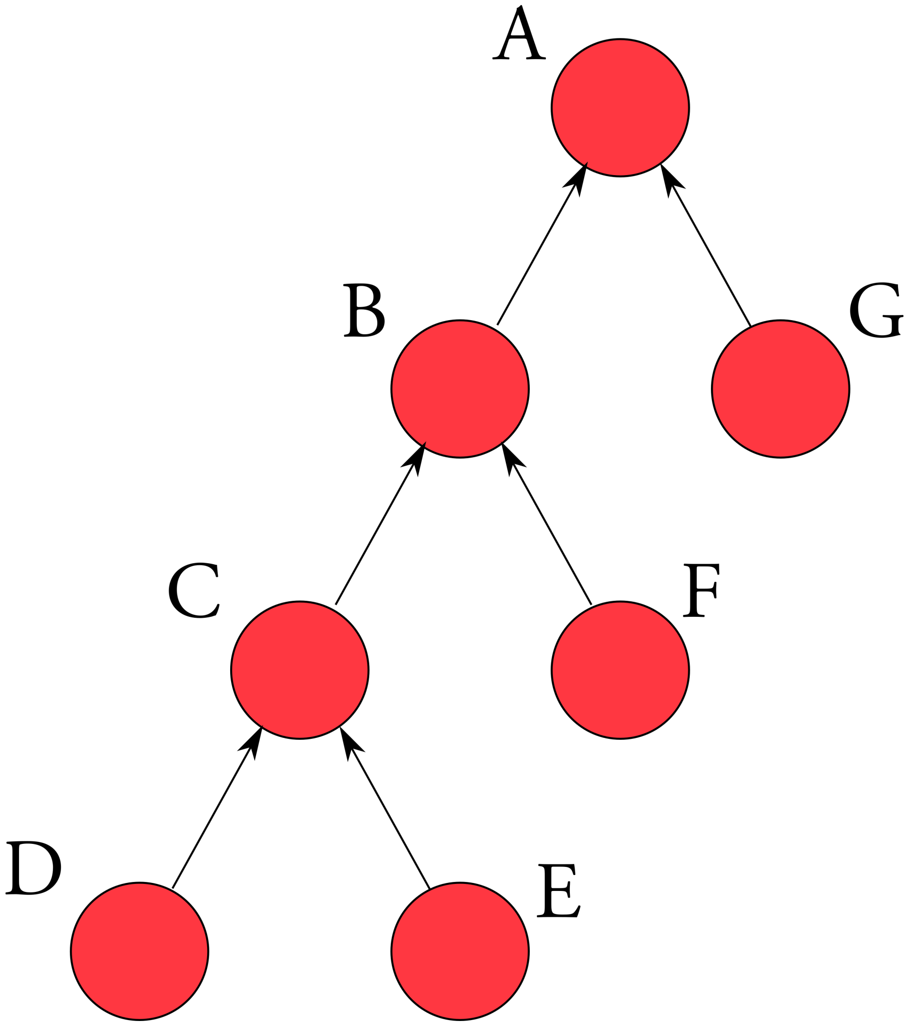
- Generate 500M floats normally distributed with average 0 and sigma 20
- Create a thread-safe histogram class with 100 bins of width 5 (first and last bins contain overflow)
- Use parallel for to push these numbers in the histogram
- Measure strong scaling
- Measure how performance changes, when modifying the number of bins
- Can you think of another pattern for mitigating high contention cases?

Parallel Scheduler

- Efficient load balancing by work stealing
- Reduce context switching
- Preserve data locality
- Keep CPUs busy
- Start/terminating tasks is up to 2 orders of magnitude faster than spawning/joining threads

Depth-first execute, breadth-first theft

- Strike when the cache is hot
  - The deepest tasks are the most recently created tasks and, therefore, the hottest in the cache
- Minimize space



Task Parallelism with TBB

- A `task_group` is a container of potentially concurrent and independent tasks
- A task can be created from a lambda or a functor
- A very stupid way to compute the Fibonacci sequence (a lot of duplicate calculations)

```
#include <iostream>
#include <oneapi/tbb.h>
#include <oneapi/tbb/task_group.h>

using namespace oneapi::tbb;

int Fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int x, y;
    task_group g;
    g.run([&] { x = Fib(n - 1); }); // spawn a task
    g.run([&] { y = Fib(n - 2); }); // spawn another task
    g.wait();                       // wait for both tasks to complete
    return x + y;
  }
}

int main() {
  std::cout << Fib(32) << std::endl;
  return 0;
}
```

backup

Parallel algorithms in C++

- Starting from C++17, parallel/vectorized versions of standard algorithms started to appear
- You mostly don't have to think about what kind of parallel implementation is hidden under the hood
- You can control the behavior by changing the execution policy

Execution Policies (since C++17)

- `std::execution::seq` : a parallel algorithm's execution may not be parallelized.
- `std::execution::par` : indicate that a parallel algorithm's execution may be executed in an unordered fashion in unspecified threads, and sequenced with respect to one another within each thread.
- `std::execution::par_unseq` : indicate that a parallel algorithm's execution may be executed in an unordered fashion in unspecified threads, and unsequenced with respect to one another within each thread.

Parallel Algorithms

- std::accumulate
- std::adjacent_difference
- std::inner_product
- std::partial_sum
- std::adjacent_find
- std::count
- std::count_if
- std::equal
- std::find
- std::find_if
- std::find_first_of
- std::for_each
- std::generate
- std::generate_n
- std::lexicographical_compare
- std::mismatch
- std::search
  std::search_n
- std::transform
- std::replace
- std::replace_if
- std::max_element
- std::merge
- std::min_element
- std::nth_element
- std::partial_sort
- std::partition
- std::random_shuffle
- std::set_union
- std::set_intersection
- std::set_symmetric_difference
- std::set_difference
- std::sort
- std::stable_sort
- std::unique_copy

Examples of what is possible

```
#include <execution>
...
std::vector<int> v;
// fill the vector
...
// sort it in parallel
std::sort(std::execution::par, v.begin(), v.end());


// apply a function foo to each element
std::for_each(std::execution::par_unseq, v.begin(), v.end(),
foo);
```

Unordered algorithms

```
std::vector<int> v;

// fill the vector

...

// reduce it in parallel

// reduction_binary_op has to be commutative and associative
 // operation

auto y = std::reduce(std::par_unseq, v.begin(), v.end(),
[initialvalue], [reduction_binary_op]);
```

`std::transform_reduce`, aka the parallel C++ swiss knife

- Takes a container of elements of type T
- Produces an object of type R
- Requires a transformation function

```
R foo(const T&)
```

- Requires a requires a binary operation:

```
R bar(const R&,const R&)
```

- Requires an initial value for the reduction

example

- The norm of a vector is:

  sqrt(x[0]*x[0] + x[1]*x[1] + ... + x[N-1]*x[N-1])

```
std::vector<double> v; // fill it

double result2 =
std::transform_reduce(std::par_unseq,
                v.begin(), v.end(),
                // transform
                [](double elt) { return elt*elt; },
                // initial value
                0.0,
                // reduction
                [](double x, double y) {return x+y;}
                );

double norm = std::sqrt(result2);
```