



High performance scientific computing in C++

HPC C++ Course 2024

28 October – 31 October 2024 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

```
1 struct Point {  
2     double x{}, y{}, z{}, w{};  
3 };
```

Assuming a cache line size of 64 bytes, and that `double` is 8 bytes, how many elements of a vector of `Point` would fit completely inside a cache line?

- (A) 1
- (B) 2
- (C) 1 or 2, can't be sure
- (D) 8

```
1 struct Point {  
2     double x{}, y{}, z{}, w{};  
3 };
```

Assuming a cache line size of 64 bytes, and that **double** is 8 bytes, how many cache lines must be read when accessing a single Point ?

- (A) 1
- (B) 2
- (C) 1 or 2, can't be sure
- (D) 8

```
1 struct alignas(32) Point {  
2     double x{}, y{}, z{}, w{};  
3 };
```

Assuming a cache line size of 64 bytes, and that `double` is 8 bytes, how many cache lines must be read when accessing a single Point ?

- (A) 1
- (B) 2
- (C) 1 or 2, can't be sure
- (D) 8

xtensor

xtensor: multi-dimensional arrays with lazy evaluation

```
1 np.linspace(0., 2., 10)
2 np.logspace(1., 10., 4)
3 np.zeros(10, 10)
4 A[1,2]
5 A.flat(4)
6 A[:,3]
7 A[:3, 3:]
8 np.vectorize(f)
9 A[A > 1.0]
10 A[[1,2], [0,1]]
11 np.random.rand(100,200)
12 np.random.shuffle(A)
13 np.where(a < 0, a , b)
14 np.loadtxt(file, delim)
15 np.linalg.svd(a)
16 np.linalg.eig(a)
```

```
1 xt::linspace<double>(0., 2., 10UL);
2 xt::logspace<double>(2., 10., 4UL);
3 xt::zeros<double>({10UL, 10UL});
4 A(1,2);
5 A[4];
6 xt::col(A, 3) or xt::view(A, xt::all, 3);
7 xt::view(A, xt::range(_, 3), xt::range(3,_));
8 xt::vectorize(f);
9 xt::filter(A, A > 1.0);
10 xt::index_view(A, {{1,2}, {0,1}});
11 xt::random::rand<double>({100, 200});
12 xt::random::shuffle(A);
13 xt::where(A < 0, A, B);
14 xt::load_csv<double>(stream);
15 xt::linalg::svd(A);
16 xt::linalg::eig(A);
```

- Syntax modelled after python numpy
- Sometimes more lazy evaluations

```
1 #include <xtensor/xtensor.hpp>
2 #include <xtensor/xarray.hpp>
3 #include <xtensor/xio.hpp>
4 #include <xtensor/xrandom.hpp>
5 #include <xtensor-blas/xlinalg.hpp>
6 #include <iostream>
7
8 auto main() -> int
9 {
10     auto R = xt::random::rand<double>({4, 4});
11     auto eigs = xt::linalg::eigvals(R);
12     std::cout << R << "\n\n";
13     std::cout << eigs << "\n";
14 }
```

Exercise 3.1:

The short program `examples/xtensor/xt0.cc` demonstrates using `xtensor` with eigenvalue evaluation. The linear algebra functionality in `xtensor` is currently handled by an external project `xtensor-blas`, which offloads some of the work to a `blas` library. To build the program, set the include path to include headers from “`xtensor-stack`”, i.e., `xtl`, `xtensor`, `xsimd`, and `xtensor-blas`. They can be given a common installation prefix. On JUSUF, the relevant include and library directories are already in the right paths. For linking, use

```
-lopenblas -lpthread -lgfortran
```

Exercise 3.2:

The program `xt1.cc` demonstrates creation of two random matrices using `xtensor`, and matrix multiplication using `xtensor-blas`. Test using compilation as above. `xt2.cc` demonstrates numerical verification that the sum of eigenvalues of a symmetric matrix is the trace of the matrix. Use these programs to familiarize yourself with `xtensor`.

Parallel computing

Parallel computing



- Engineering (power consumption) challenges make processors with higher and higher clock rates impractical
- Computers in the last 20 years have instead increased processing power by adding more hardware for parallel processing

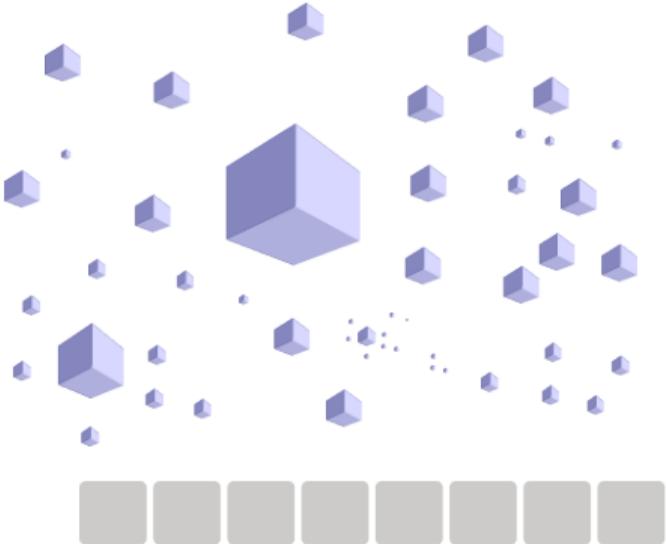
Parallel computing

```
1 auto gcd(unsigned s, unsigned l) -> unsigned
2 {
3     if (s > l)
4         std::swap(s, l);
5     while (s != 0) {
6         auto r = l % s;
7         l = s;
8         s = r;
9     }
10    return l;
11 }
```



- Engineering (power consumption) challenges make processors with higher and higher clock rates impractical
- Computers in the last 20 years have instead increased processing power by adding more hardware for parallel processing
- A sequence of dependent operations on a small set of entities is ill-suited for processing with many workers

Parallel computing



- Engineering (power consumption) challenges make processors with higher and higher clock rates impractical
- Computers in the last 20 years have instead increased processing power by adding more hardware for parallel processing
- A sequence of dependent operations on a small set of entities is ill-suited for processing with many workers
- Given a large amount of information to be processed, or a task with a large number of independent sub-tasks, it is possible to reduce the overall processing time.

Parallel computing

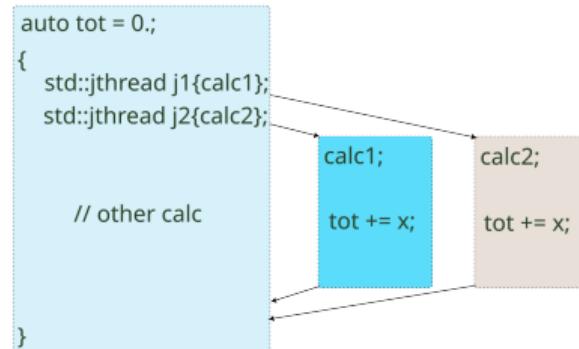
What mechanisms do we have in C++ to exploit available parallelism in hardware?

- Threads, mutexes, atomic operations
- RAII for resource management
- Libraries to partition and assign work to workers
- Templates, lambda functions, CTAD
- High-level STL style algorithms abstracting common programming building blocks
- Containers and allocators for more efficient (and correct) parallel processing

Threads

```
1 auto calc1 = [=] () {
2     auto tot1 = 0.;
3     for (auto i = 0UL; i < N; ++i) {
4         auto ang = 2 * i * pi / N;
5         tot1 += std::cos(ang) * std::cos(ang);
6     }
7 };
8 auto calc2 = [=] () {
9     auto tot1 = 0.;
10    for (auto i = 0UL; i < N; ++i) {
11        auto ang = 2 * i * pi / N;
12        tot1 += std::sin(ang) * std::sin(ang);
13    }
14 };
15 std::jthread j1 { calc1 };
16 std::jthread j2 { calc2 };
```

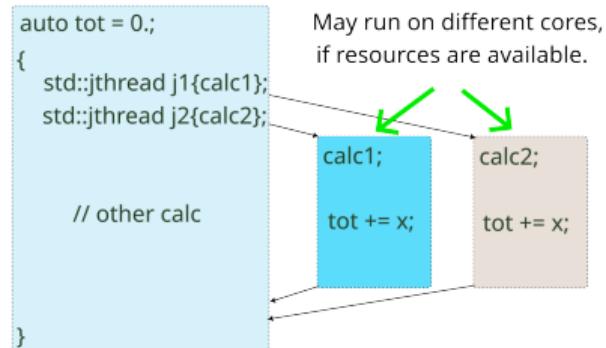
- `std::thread`, `std::async` ... since C++11
- Parallel algorithms since C++17
- `std::jthread`, `std::stop_token` since C++20
- `std::jthread` joins in the destructor



Threads

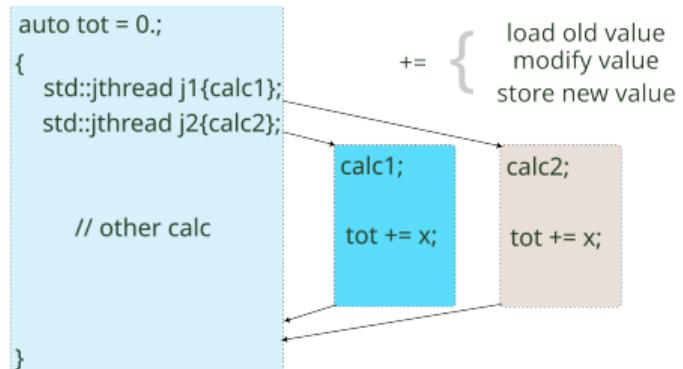
```
1 auto calc1 = []() {
2     auto tot1 = 0.;
3     for (auto i = 0UL; i < N; ++i) {
4         auto ang = 2 * i * pi / N;
5         tot1 += std::cos(ang) * std::cos(ang);
6     }
7 };
8 auto calc2 = []() {
9     auto tot1 = 0.;
10    for (auto i = 0UL; i < N; ++i) {
11        auto ang = 2 * i * pi / N;
12        tot1 += std::sin(ang) * std::sin(ang);
13    }
14 };
15 std::jthread j1 { calc1 };
16 std::jthread j2 { calc2 };
```

- `std::thread`, `std::async` ... since C++11
- Parallel algorithms since C++17
- `std::jthread`, `std::stop_token` since C++20
- `std::jthread` joins in the destructor



Threads

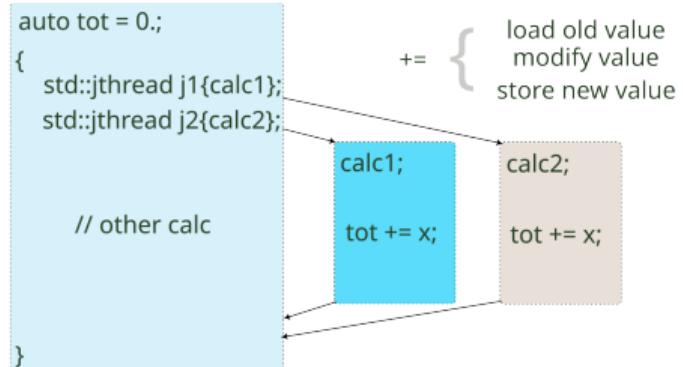
```
1 auto tot = 0.;  
2 {  
3     std::jthread j1 { [&]() {  
4         for (auto i = 0UL; i < N; ++i) {  
5             auto ang = 2 * i * pi / N;  
6             tot += std::cos(ang) * std::cos(ang);  
7         }  
8     };  
9     std::jthread j2 { [&]() {  
10         for (auto i = 0UL; i < N; ++i) {  
11             auto ang = 2 * i * pi / N;  
12             tot += std::sin(ang) * std::sin(ang);  
13         }  
14     };  
15 };  
16 std::cout << "Total " << tot << "\n";
```



- Modification of data at the same address from multiple threads can lead to “data races”

Threads

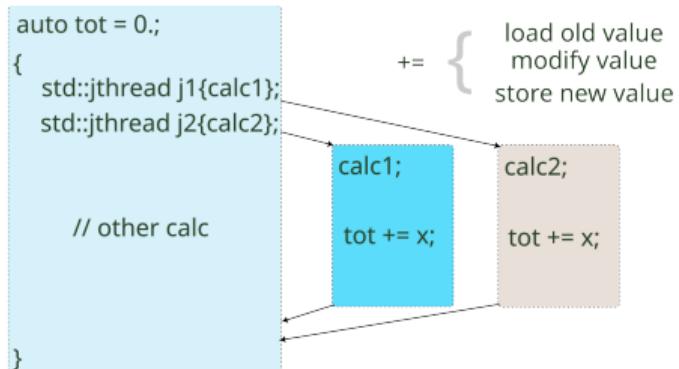
```
1 auto tot = 0.;  
2 {  
3     std::jthread j1 { [&]() {  
4         for (auto i = 0UL; i < N; ++i) {  
5             auto ang = 2 * i * pi / N;  
6             tot += std::cos(ang) * std::cos(ang);  
7         }  
8     };  
9     std::jthread j2 { [&]() {  
10         for (auto i = 0UL; i < N; ++i) {  
11             auto ang = 2 * i * pi / N;  
12             tot += std::sin(ang) * std::sin(ang);  
13         }  
14     };  
15 };  
16 std::cout << "Total " << tot << "\n";
```



- The result can be incorrect, since the load-modify-commit operations from the two threads can overlap

Threads

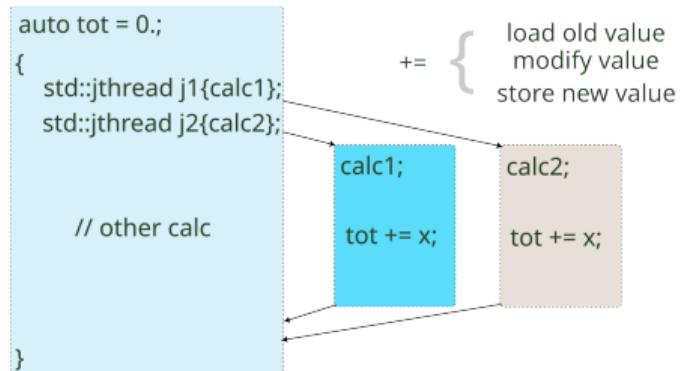
```
1 std::mutex totmutex;
2
3     std::jthread j1 { [&]() {
4         for (auto i = 0UL; i < N; ++i) {
5             auto ang = 2 * i * pi / N;
6             std::scoped_lock lck { totmutex };
7             tot += std::cos(ang) * std::cos(ang);
8         }
9     } };
10    std::jthread j2 { [&]() {
11        for (auto i = 0UL; i < N; ++i) {
12            auto ang = 2 * i * pi / N;
13            std::scoped_lock lck { totmutex };
14            tot += std::sin(ang) * std::sin(ang);
15        }
16    } };
17 }
18 std::cout << "Total " << tot << "\n";
```



- Fix 1: `std::mutex`: A resource which can be acquired by only one thread at a time. Must be released by the acquiring thread.
`std::scoped_lock` manages mutex acquisition/release using RAII

Threads

```
1 std::atomic<double> tot {};
2 {
3     std::jthread j1 { [&]() {
4         for (auto i = 0UL; i < N; ++i) {
5             auto ang = 2 * i * pi / N;
6             tot += std::cos(ang) * std::cos(ang);
7         }
8     } };
9     std::jthread j2 { [&]() {
10         for (auto i = 0UL; i < N; ++i) {
11             auto ang = 2 * i * pi / N;
12             tot += std::sin(ang) * std::sin(ang);
13         }
14     } };
15 }
16 std::cout << "Total " << tot << "\n";
```



- `std::atomic<T>` gives us “atomic” load-modify-commit operations

Threads

```
1 struct wrapped1 {
2     int val {};
3 };
4 template <class W>
5 struct func {
6     void operator() (volatile W* var)
7     {
8         for (unsigned i = 0; i < WORKLOAD / PARALLEL; ++i) {
9             var->val = var->val + 1;
10        }
11    }
12 };
13 {
14     std::array<wrapped2, PARALLEL> arr {};
15     {
16         std::array<std::jthread, PARALLEL> threads;
17         for (unsigned i = 0U; i < PARALLEL; ++i) {
18             threads[i] =
19                 std::jthread(func<wrapped2>{}, &arr[i]);
20         }
21     }
22 }
```

- Even when threads write to different addresses, there can be a significant slowdown because of “false sharing”

Threads

```
1 struct align_as(std::hardware_destructive_interference_size)
2 wrapped1 {
3     int val {};
4 };
5 template <class W>
6 struct func {
7     void operator() (volatile W* var)
8 {
9     for (unsigned i = 0; i < WORKLOAD / PARALLEL; ++i) {
10         var->val = var->val + 1;
11     }
12 }
13 };
14 {
15     std::array<wrapped2, PARALLEL> arr {};
16 {
17     std::array<std::jthread, PARALLEL> threads;
18     for (unsigned i = 0U; i < PARALLEL; ++i) {
19         threads[i] =
20             std::jthread(func<wrapped2>{}, &arr[i]);
21     }
22 }
23 }
```

- Even when threads write to different addresses, there can be a significant slowdown because of “false sharing”
- Mitigation: alignment or padding

Parallel STL

Parallel STL

- Parallel versions of the high-level building blocks such as `std::sort`, `std::reduce` etc.
- C++17 parallel STL provides a way to express that something can be done in parallel, but does not mandate implementation strategy
- Programs already written using algorithms will offer many opportunities for exploiting parallelism
- A TBB based implementation is used since GCC 9.1. Intel and Microsoft compilers have their implementations as well.
- `std::sort` sorts.
`std::sort(std::execution::par, ...)` sorts in parallel
- `std::reduce` adds up elements from a range.
`std::reduce(std::execution::par, ...)` adds up elements in parallel

```
1 std::sort(std::execution::par,
2           points.begin(), points.end(),
3           [] (auto p1, auto p2) {
4               return p1.x() < p2.x();
5           });
6 std::for_each(std::execution::par_unseq,
7               points.begin(), points.end(),
8               [] (auto & p) {
9                   p.norm(1);
10                });
11 }
```

- As of GCC 14.2, to compile programs using parallel algorithms, we need to link with `libtbb` and `libtbbmalloc`, e.g.,
`G par_user.cc -ltbb -ltbbmalloc`
- As of Clang 19.1, parallel STL remains an **experimental** feature in `libc++`, and must be enabled through `-fexperimental-library`

Execution policies

- `std::execution::sequenced_policy` : Parallel algorithm's execution may not be parallelised. Element wise operations are indeterminately sequenced in the calling thread. An instance called, `std::execution::seq` is usually used to disambiguate overload resolution
- `std::execution::parallel_policy` : May be parallelised. Element wise operations can happen in the calling thread, or on another. Relative sequencing is indeterminate. Convenience instance:
`std::execution::par`
- `std::execution::parallel_unsequenced_policy` May be parallelised and vectorised. Element wise operations can run in unspecified threads, and can be unordered in each thread.
`std::execution::par_unseq`
- `std::execution::unsequenced_policy` Only vectorised. `std::execution::unseq`

Parallel STL examples

Exercise 3.3:

The program `examples/pstl/inner_product.cc` demonstrates the use of the parallel STL library, performing a simple inner product calculation. Use `-ltbb -ltbbmalloc` for linking, or use the CMake file in the directory.

Exercise 3.4:

The program `examples/pstl/transform_reduce.cc` creates a vector of random points in 2D, and then calculates the moment of inertia using STL algorithms. Just switching the execution policy parameter, the program can be parallelised and vectorised. Test!

Parallel STL examples

Exercise 3.5:

Parallelise the program `exercises/pstl/mandelbrot0.cc` using parallel STL.

Exercise 3.6:

At what size of a group of random strangers does the chance of two people sharing a birthday become greater than 0.5? The program `birthday_problem.cc` solves it using a crude, brute force Monte Carlo simulation. Parallelise it using parallel STL.

Examples in this section can be done with both GCC and Clang, with some caveats when using Clang.

`clang++ -std=c++23 -stdlib=libc++ -fexperimental-library -O3 -march=native ____.cc`
and

`clang++ -std=c++23 -stdlib=libstdc++ -O3 -march=native ____.cc`

will both will work. As of October 2024, `libc++` has not optimised performance when using parallel algorithms.

Threading Building Blocks

TBB: Threading Building Blocks I

- Provides utilities like `parallel_for`, `parallel_reduce` to simplify the most commonly used structures in parallel programs
- Provides scalable concurrent containers such as vectors, hash tables and queues for use in multi-threaded environments
- No direct support for vector parallelism. But can be combined with auto-parallelisation and `#pragma omp simd` etc or explicit SIMD with a SIMD library
- Supports complex models such as pipelines, data flow and unstructured task graphs
- Scalable memory allocation, avoidance of false sharing, thread local storage
- Low level synchronisation tools like mutexes and atomics
- Work stealing task scheduler
- <http://www.threadingbuildingblocks.org>
- **Structured Parallel Programming**, Michael McCool, Arch D. Robinson, James Reinders

Using TBB

- Public names are available under the namespaces `tbb` and `tbb::flow`
- You indicate "available parallelism", scheduler may run it in parallel if resources are available
- Unnecessary parallelism will be ignored

parallel invoke

```
1 void prep(Population &p);
2 void iomanage();
3 tbb::parallel_invoke(
4     [&] {
5         noise_w(0., pars.sigma, wns);
6         std::copy(wns.begin(), wns.end(), wnoisemat.begin());
7     },
8     [&] {
9         noise_phi(0., pars.sigma, phins);
10        std::copy(phins.begin(), phins.end(), phinoisemat.begin());
11    });

```

Exercise 3.7: examples/tbb/parallel_invoke.cc

Compile with

```
G parallel_invoke.cc -ltbb -ltbbmalloc
```

- A few adhoc tasks which do not depend on each other
- Runs them in parallel
- waits until all of them are finished

TBB task groups

```
1 struct Equation {
2     void solve();
3 };
4
5 std::list<Equation> equations;
6 tbb::task_group g;
7 for (auto eq : equations)
8     g.run([]{eq.solve();});
9
10 g.wait();
```

- Run an arbitrary number of callable objects in parallel
- In case an exception is thrown, the task group is cancelled

TBB task arena

```
1 auto main(int argc, char *argv[]) -> int
2 {
3     size_t nthreads=std::stoul(argv[1]);
4     tbb::task_arena main_executor;
5     main_executor.initialize(nthreads);
6     main_executor.execute([&]{
7         haha();
8     });
9 }
10 void haha()
11 {
12     ...
13     tbb::parallel_invoke(a,b,c,d,e);
14 }
15 void a()
16 {
17     tbb::parallel_for(...);
18 }
```

- Task arena to manage tasks, maps them to threads etc.
- Number of threads in an arena limited by its concurrency level
- Execute function, with a function object as argument.
- Returns the same thing as the function it is executing.

Parallel for loops

- Template function modelled after the `for` loops, like many STL algorithms
- Takes a `callable object` as the third argument
- Using lambda functions, you can expose parallelism in sections of your code

```
1   tbb::parallel_for(first, last, f);
2   // parallel equivalent of
3   // for (auto i=first; i<last; ++i) f(i);
4
5   tbb::parallel_for(first, last, stride, f);
6   // parallel equivalent of
7   // for (auto i=first; i<last; i+=stride)
8   //     f(i);
9
10  tbb::parallel_for(first, last,
11                  [captures](anything) {
12                  //Code that can run in parallel
13              });
14
```

Parallel for with ranges

- Splits `range` into smaller ranges, and applies `f` to them in parallel
- Possible to optimize `f` for sub-ranges rather than a single index
- Any type satisfying a few design conditions can be used as a range
- Multidimensional ranges possible

```
1 tbb::parallel_for(0,1000000,f);
2 // One parallel invocation for each i!
3 tbb::parallel_for(range,f);
4
5 // A type R can be a range if the
6 // following are available
7 R::R(const R &);
8 R::~R();
9 bool R::is_divisible() const;
10 bool R::empty() const;
11 R::R(R & r,split); //Split constructor
```

Parallel for with ranges

```
1 tbb::blocked_range<int> r{0,30,20};  
2 assert(r.is_divisible());  
3 blocked_range<int> s{r};  
4 //Splitting constructor  
5 assert(!r.is_divisible());  
6 assert(!s.is_divisible());  
7
```

- `tbb::blocked_range<int>(0, 4)` represents an integer range 0..4
- `tbb::blocked_range<int>(0, 50, 30)` represents two ranges, 0..25 and 26..50
 - So long as the size of the range is bigger than the "grain size" (third argument), the range is split

Parallel for with ranges

```
1 void dasxpcy_tbb(double a, std::span<const double> x, std::span<double> y) {
2     tbb::parallel_for(tbb::blocked_range<int>(0, x.size()),
3                         [&](tbb::blocked_range<int> r) {
4                 for (size_t i = r.begin(); i != r.end(); ++i) {
5                     y[i] = a * sin(x[i]) + cos(y[i]);
6                 }
7             });
8 }
```

- `parallel_for` with a range uses split constructor to split the range as far as possible, and then calls `f(range)`, where `f` is the functional given to `parallel_for`
- It is unlikely that you wrote your useful functions with ranges compatible with `parallel_for` as arguments
- But with lambda functions, it is easy to fit the parts!

Exercise 3.8: TBB parallel for demo

The program `examples/dasxpcy.cc` demonstrates the use of parallel for in TBB. It is a slightly modified version of the commonly used DAXPY demos. Instead of calculating $y = a * x + y$ for scalar a and large vectors x and y , we calculate $y = a * \sin(x) + \cos(y)$. To compile, you need to load your compiler and TBB modules, and use them like this:

```
1 G dasxpcy.cc -ltbb -ltbbmalloc
```

2D ranges

```
1 void f(size_t i, size_t j);
2 tbb::blocked_range2d<size_t> r{0, N, 0, N};
3 tbb::parallel_for(r, [&](tbb::blocked_range2d<size_t> r) {
4     for (auto i = r.rows().begin(); i != r.rows().end(); ++i) {
5         for (auto j = r.cols().begin(); j != r.cols().end(); ++j) {
6             f(i, j);
7         }
8     }
9});
```

- `rows()` is an object with a `begin()` and an `end()` returning just the integer row values in the range.
Similarly: `cols()` ...
- 2D range can also be split
- The callable object argument should assume that the original 2D range has been split many times, and we are operating on a smaller range, whose properties can be accessed with these functions.

Parallel reductions with ranges

```
1 T result = tbb::parallel_reduce(range, identity, subrange_reduction, combine);
```

- `range` : As with parallel for
- `identity` : Identity element of type T. The type determines the type used to accumulate the result
- `subrange_reduction` : Functor taking a "subrange" and an initial value, returning reduction
- `combine` : Functor taking two arguments of type T and returning reduction over them over the subrange.
Must be associative, but not necessarily commutative.

Parallel reduce with ranges

```
1 double inner_prod_tbb(std::span<const double> x, std::span<double> y) {
2     return tbb::parallel_reduce(
3         tbb::blocked_range<int>(0, n), // range
4         double{}, // identity
5         [&](tbb::blocked_range<int> &r, float in) {
6             return std::inner_product(x.begin() + r.begin(), x.begin() + r.end(),
7                                       y.begin() + r.begin(), in);
8         }, // subrange reduction
9         std::plus<double>{} // combine
10    );
11 }
```

- With TBB ranges, we can use blocked implementations with hopefully vectorisable calculations in subranges
- Two functors are required, either of which could be lambda functions
- Important to add the contribution of initial value in subrange reductions

Exercise 3.9: TBB parallel reduce

The program `tbbreduce.cc` is a demo program to calculate an integral using `tbb::parallel_reduce`. What kind of speed up do you see relative to the serial version ? Does it make sense considering the number of physical cores in your computer ?

Atomic variables

- "Instantaneous" updates
- Lock-free synchronization
- For `std::atomic<T>`, T can be integral, enum or pointer type, and since C++20, also floating point, `std::shared_ptr` and `std::weak_ptr`
- If `index.load() == k` simultaneous calls to `index++` by n threads will increase `index` to `k + n`. Each thread will use a distinct value between `k` and `k + n`

```
1 std::array<double, N> A;
2 std::atomic<int> index;
3
4 void append(double val)
5 {
6     A[index++] = val;
7 }
```

Atomic variables

- "Instantaneous" updates
- Lock-free synchronization
- For `std::atomic<T>`, T can be integral, enum or pointer type, and since C++20, also floating point, `std::shared_ptr` and `std::weak_ptr`
- If `index.load() == k` simultaneous calls to `index++` by n threads will increase `index` to `k + n`. Each thread will use a distinct value between `k` and `k + n`

```
1 std::array<double, N> A;
2 std::atomic<int> index;
3
4 void append(double val)
5 {
6     A[index++] = val;
7 }
```

But it is important that we use the return value of `index++` in the threads!

Enumerable thread specific

```
1 tbb::enumerable_thread_specific<double> E;
2 double Eglob=0;
3 double f(size_t i, size_t j);
4 tbb::blocked_range2d<size_t> r{0, N, 0, N};
5 tbb::parallel_for(r, [&](tbb::blocked_range2d<size_t> r) {
6     auto & eloc = E.local();
7     for (size_t i = r.rows().begin(); i != r.rows().end(); ++i) {
8         for (size_t j = r.cols().begin(); j != r.cols().end(); ++j) {
9             if (j > i) eloc += f(i, j);
10        }
11    }
12 });
13 Eglob = 0;
14 for (auto& v : E) {Eglob += v; v = 0;}
```

- Thread local "views" of a variable
- behaves like an STL container of those views
- Member function `local()` gives a reference to the local view in the current thread
- Any thread can access all views by treating it as an STL container

TBB allocators

- Dynamic memory allocation in a multithreaded program must avoid conflicts from `new` calls from different threads
- Global memory lock

TBB allocators

- Interface like `std::allocator`, so that it can be used with STL containers. E.g.,
`std::vector<T, tbb::cache_aligned_allocator<T>>`
- `tbb::scalable_allocator<T>` : general purpose scalable allocator type, for rapid allocation from multiple threads
- `tbb::cache_aligned_allocator<T>` : Allocates with cache line alignment. As a consequence, objects allocated in different threads are guaranteed to be in different cache lines.

Concurrent containers

```
1 #include <tbb/concurrent_vector.h>
2
3 auto v = tbb::concurrent_vector<int>(N, 0);
4
5 tbb::parallel_for(v.range(), [&](tbb::concurrent_vector::range_type r) {
6     //...
7 });
}
```

- Random access by index
- Multiple threads can grow container and add elements concurrently
- Growing the container does not invalidate any iterators or indexes
- Has a `range()` member function for use with `parallel_for` etc.

Linear Algebra

Linear algebra

- Operations on matrices, vectors, linear systems etc.
- Data parallel, simple numerical calculations
- Can be hand coded, but taking proper account of available CPU instructions, memory hierarchy etc is hard
- Libraries with standardized syntax for wide applicability
- Excellent vendor libraries are available on HPC systems

Eigen: A C++ template library for linear algebra

- Include only library. Download from

<http://eigen.tuxfamily.org/>, unpack in a location of your choice, and use. Nothing to link.

- Small fixed size to large dense/sparse matrices
- Matrix operations, numerical solvers, tensors ...
- Expression templates: lazy evaluation, smart removal of temporaries

```
1 // examples/Eigen/eigen1.cc
2 #include <iostream>
3 #include <Eigen/Dense>
4 using namespace Eigen;
5 using namespace std;
6 int main()
7 {
8     MatrixXd m=MatrixXd::Random(3, 3);
9     m = (m + MatrixXd::Constant(3, 3, 1.2)) * 50;
10    cout << "m =" << "\n" << m << "\n";
11    VectorXd v(3);
12    v << 1, 2, 3;
13    cout << "m * v =" << "\n" << m * v << "\n";
14 }
```

G eigen1.cc

- Explicit vectorization
- Elegant API

Eigen: matrix types

- `MatrixXd` : matrix of arbitrary dimensions
- `Matrix3d` : fixed size 3×3 matrix
- `Vector3d` : fixed size 3d vector
- Element access `m(i, j)`
- Output `std::cout << m << "\n";`
- Constant : `MatrixXd::Constant(a, b, c)`
- Random : `MatrixXd::Random(n, n)`
- Products : `m * v` or `m1 * m2`
- Expressions : `3 * m * m * v1 + u * v2 + m * m * m * v3`
- Column major matrix : `Matrix<float, 3, 10, Eigen::ColMajor>`

Eigen: matrix operations

```
1 #include <iostream>
2 #include <Eigen/Dense>
3 using namespace Eigen;
4 auto main() -> int {
5     Matrix3f A;
6     Vector3f b;
7     A << 1,2,3, 4,5,6, 7,8,10;
8     b << 3, 3, 4;
9     std::cout << "Here is the matrix A:\n" << A << "\n";
10    std::cout << "Here is the vector b:\n" << b << "\n";
11    Vector3f x = A.colPivHouseholderQr().solve(b);
12    std::cout << "The solution is:\n" << x << "\n";
13 }
```

- Blocks `m.block(start_r, start_c, nr, nc)`, or `m.block<nr,nc>(start_r, start_c)`
-

```
1 SelfAdjointEigenSolver<Matrix2f> eigensolver(A);
2 if (eigensolver.info() != Success) abort();
3 std::cout << "Eigenvalues " << eigensolver.eigenvalues() << "\n";
```

Eigen: examples

Exercise 3.10:

There are a few example programs using Eigen in the folder `examples/Eigen`. Read the programs `eigen0.cc` and `eigen1.cc`. To compile, use `G program.cc`.

Exercise 3.11:

The folder `examples/Eigen` contains a matrix multiplication example, `matmul.cc` using Eigen. Compare with a naive version of a matrix multiplication program, `matmul_naive.cc`, by compiling and running both programs. Try different matrix sizes. Then, you can use a parallel version of the Eigen matrix multiplication by recompiling with `-fopenmp`.

Exercise 3.12:

The file `exercises/PCA` has a data file with tabular data. Each column represents all measurements of a particular type, while each row is a different trial. In each row, the first column, x_{i0} , represents a pseudo-time variable. Write a program using Eigen to perform a Principal Component Analysis on this data set, ignoring the first column. Hint:

if $X_i = [x_{i1}, x_{i2}, \dots x_{im}]$ is the data of row i , the covariance matrix is defined as,

$$C_{ab} = \frac{1}{(n - 1)} \sum_k x_{ka} x_{kb}$$

The principal components of the data are obtained by right multiplying the data matrix by the matrix whose columns are the eigen vectors of the matrix C_{ab} , conventionally ordered by decreasing eigenvalues.

Lessons from matrix multiplication

Exercise 3.13:

In the examples folder, you will find a `MatMul` subfolder, containing a written lesson called `SessionMatrix.pdf`. This file contains 8 stages organised as exercises starting with a naive implementation of a matrix type in C++, and ending with something with reasonably respectable performance (comparable to what is possible with, e.g., Eigen, or other BLAS libraries) on a single node on JUSUF. It only uses concepts introduced in this course, and does not call any linear algebra library function. Work through the exercises and test the different stages on JUSUF!