HIGH PERFORMANCE SCIENTIFIC COMPUTING IN C++

PATC HPC C++ course 2021

21–24 June 2021 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

xtensor

xtensor: multi-dimensional arrays with lazy evaluation

```python
np.linspace(0., 2., 10)
np.logspace(1., 10., 4)
np.zeros(10, 10)
A[1,2]
A.flat(4)
A[:,3]
A[:3, 3:]
np.vectorize(f)
A[A > 1.0]
A[[1,2], [0,1]]
np.random.rand(100,200)
np.random.shuffle(A)
np.where(a < 0, a , b)
np.load_txt(file, delim)
np.linalg.svd(a)
np.linalg.eig(a)
```

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

- Syntax modelled after python numpy
- Sometimes more lazy evaluations

```cpp
#include <xtensor/xtensor.hpp>
#include <xtensor/xarray.hpp>
#include <xtensor/xio.hpp>
#include <xtensor/xrandom.hpp>
#include <xtensor-blas/xlinalg.hpp>
#include <iostream>

auto main() -> int
{
    auto R = xt::random::rand<double>({4, 4});
    auto eigs = xt::linalg::eigvals(R);
    std::cout << R << "\n\n";
    std::cout << eigs << "\n";
}
```

Exercise 3.1:

The short program `examples/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`, `xtensor-io` 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`

Parallel STL

Parallel STL

- Parallel versions of STL algorithms: C++17
- Specifies the operations to be performed in the STL style abstract way, but does not mandate implementation strategy
- Programs already written using algorithms will offer many opportunities for introducing 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               });
```

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 3.2:

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 3.3:

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 ptogram can be parallelised and vectorised. Test!

Exercise 3.4:

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

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 3.5: examples/tbb/parallel_invoke.cc

Compile with `G -tbb parallel_invoke.cc`

- 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 });
```

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());
```

- `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::vector<double> &x, std::vector<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 3.6: TBB parallel for demo

The program `examples/daxpcy.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 -tbb daxpcy.cc
```

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 (size_t i=r.rows().begin();i!=r.rows().end();++i) {
5          for (size_t 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::vector<double> & x, std::vector<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 3.7: 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  }
```

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.

Lessons from matrix multiplication

Exercise 3.8:

In the example 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 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!

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 -eigen eigen1.cc
```

- Explicit vectorization
- Elegant API

Eigen: matrix types

- `MatirxXd` : 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

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

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

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

Eigen: examples

Exercise 3.9:

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 -eigen program.cc`.

Exercise 3.10:

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 3.11:

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 $X_i = [x_{i1}, x_{i2}, ...x_{im}]$ 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.