Lab 05

PBS and C++17 STD algorithms introduction

Paolo Joseph Baioni
paolojoseph.baioni@polimi.it

November 7, 2025

First introduction to PBS

OpenPBS is a software which optimises job scheduling and workload management in high-performance computing environments.
First steps:

- set up ssh on the cluster
- set up pbs environment: `. /etc/profile.d/pbs.sh`

The second step is actually optional, you can also avoid it and use the full path, eg
`/opt/pbs/bin/pbsnodes -aSj`

First introduction to PBS

The main commands we will use are:

- `pbsnodes -aSj` to check available nodes
- `qstat`, to interrogate queues
- `qsub`, to submit jobs
- `qdel`, to delete submitted jobs

In particular, the base submission command is `qsub -I` (capital i), which requests an interactive job with default options (ncpus=1, first available cpu); ctrl+D to exit.

First introduction to PBS

The powerful feature of qsub is the ability to submit batch jobs via submission scripts; you can find some hello world examples in the repo.
It's worth noting that the same options which can be specified via script can also be used CLI for an interactive session, so, for example `qsub -I -l select=1:ncpus=2:host=cpu03` will ask for an interactive job on a single node, 2 cpus (threads) on the host with short hostname cpu03; parameters not specified, such as queue and walltime, get default values.

A dive into std::algorithms & execution policies

Ingredients:

- lambdas & function objects
- C++17 standard algorithms and execution policies
- gcc toolchain and tbb module (or equivalent in the course docker/podman container)
- extra: ad-hoc compilers for offloading execution to devices (eg GPU, but works also on CPU), such as nvhpc container*.

*Can be built with:

```
sudo docker run --gpus all -it --rm nvcr.io/nvidia/nvhpc:25.9-devel-cuda_multi-ubuntu24.04
```

or

```
singularity build nvhpc.sif docker://nvcr.io/nvidia/nvhpc:25.9-devel-cuda_multi-ubuntu24.04
```

Requires proper drivers for GPU execution.
It takes some minutes to download, so, if you want to use it on your own laptop, it might be better to start the build now

Remark: With C++17, the parallel version requires to link against the Intel Threading Building Blocks (`TBB`) library (in the mk modules, preprocessor flags `-I${mkTbbInc}`, linker flags `-L${mkTbbLib} -ltbb`) .

Lambdas and Functors recap

1. Lambdas are un-named function objects

```
vector<real_t> v = {0, 1, 2, 3};
real_t a = 4.;
auto f = [&v,a](idx_t i) { return v[i] * a; };
cout << "4 = " << f(1) << endl;

struct __unnamed
{
 real_t s;
 vector <real_t>& v;
 real_t operator()(int i) { return v[i] * a;}
};
__unnamed f{a, v};
cout << "4 = " << f(1) << endl;
```

2. You can pass vectors also copying pointers (gpu safe, see last slides)

```
vector<real_t> v = {0, 1, 2, 3};
real_t a = 4.;
auto f = [v = v.data (), a](idx_t i) { return v[i] * a; };
cout << "4 = " << f(1) << endl;
```

C++17 Algorithms Recap

From for loops

```
vector <real_t> u (3), v = {0, 1, 2};
for (idx_t i = 0; i < u.size () ; ++i)
  u[i] =  v[i];
```

to std::algorithm

```
transform (v.cbegin (), v.cend (), u.begin (), [](const real_t & vi){return vi;});
```

That takes an optional `ExecutionPolicy&& policy` as first parameter

C++17 - C++20 Execution Policies - I

The execution policy types:

```
std::execution::sequenced_policy
std::execution::parallel_policy
std::execution::parallel_unsequenced_policy
std::execution::unsequenced_policy
```

have the following respective instances:

```
std::execution::seq,
std::execution::par,
std::execution::par_unseq, and
std::execution::unseq.
```

These instances are used to specify the execution policy of parallel algorithms, i.e., the kinds of parallelism allowed.

C++17 - C++20 Execution Policies - II

1. **seq**: requires that a parallel algorithm's execution may not be parallelized.
2. **par**: indicates that a parallel algorithm's execution may be parallelized. The invocations of element access functions in parallel algorithms invoked with this policy are permitted to execute in either the invoking thread or in a thread implicitly created by the library to support parallel algorithm execution.
3. **par_unseq**: indicates that a parallel algorithm's execution may be parallelized, vectorized, or migrated across threads. The invocations of element access functions in parallel algorithms invoked with this policy are permitted to execute in an unordered fashion in unspecified threads, and unsequenced with respect to one another within each thread.
4. **unseq** (C++20): indicates that a parallel algorithm's execution may be vectorized, e.g., executed on a single thread using instructions that operate on multiple data items.

Remark When using any parallel execution policy, it is the programmer's responsibility to avoid data races and deadlocks.

Indexing

In for loops we can use plain indices; given some functional `f` and vector v, we have

```
for (idx_t i = 0; i < v.size (); ++i)
    v[i] = f(i);
```

In std::algorithms we can use:

- raw pointers (error prone, not suggested)
- index vectors (memory inefficient, not suggested)
- counting iterators (external library required)
- C++20 std::ranges and std::views (full support from C++23, eg, for n-dimensional arrays)

Raw pointers

Compute index as difference between each element memory address and raw pointer to vector data

```
std::transform (v.begin (), v.end (),
                [v = v.data(), f](real_t vi)
                {
                  auto i = &vi - v;
                  return f(i);
                });
```

Vectors of indices

Easiest solution, highly inefficient

```
std::vector <idx_t> indices (v.size ());
std::iota (indices.begin (), indices.end (), 0);
// now indices is {0,1,2...v.size()-1}
std::for_each (indices.begin (), indices.end (),
               [v = v.data()](idx_t i)
               {
                 v[i] = f(i);
               });
```

Almost equal to the traditional for loop: ease of porting, but we are allocating a full vector of indices

Remark Avoid using vector of indices, it is shown here only for didactic purposes, to help giving an operational definition of counting iterators and views in this realm.

Counting iterators

One of the best available solutions, requires:

- C++17
- a library (boost and thrust are header only, it is enough to include them; boost is in the mk modules)

```
boost::counting_iterator <idx_t> first(0), last(v.size ());
std::for_each (first, last,
               [v = v.data(), f](idx_t i)
               {
                 v[i] = f(i);
               });
```

Remark: compile with `g++ -std=c++20 -I$mkBoostInc boost_example.cpp`

Ranges and views

One of the best available solutions; features:

- C++20 required, 23 for full support
- no external dependencies needed

C++20:

```
auto indices = std::views::iota (0, v.size ());
std::for_each (indices.begin (), indices.end (),
              [v = v.data()](idx_t i)
              {
                v[i] = f(i);
              });
```

From C++23*

```
std::views::cartesian_product(std::views::iota (0, m), std::views::iota (0, n));
...
```

Remark: remember to `#include <ranges>` for the definition of `std::views::iota`

*In previous standards you could rely on proposals for missing features, eg https://github.com/ericniebler/range-v3, but, in case, it becomes competitive with boost

Example application: Single/Double precision A $\times$ X + Y

SAXPY and DAXPY stands for Single and Double-Precision A $\times$ X + Y
It is a basic operation in linear algebra and is commonly used in numerical computing.
We will use it to show examples of:

- basic std algoritghms usage
- efficient std algorithms usage
- parallel std algorithms usage and performance metrics (bandwidth)
- possible effects of floats precision

Saxpy - slow

Using STL defined std::algorithms only

```
void saxpy_slow (float a, std::vector <float> & x, std::vector <float> & y)
{
  std::vector <float> temp (x.size ());
  std::fill (temp.begin (), temp.end (), a);
  std::transform (x.begin (), x.end (), temp.begin (), temp.begin (), std::
      multiplies <float> ());
  std::transform (temp.begin (), temp.end (), y.begin (), y.begin (), thrust::plus
       <float> ());
}
```

Saxpy - fast - full example I

Using user-defined function object and `std::for_each`

```
#include <algorithm>
#include <iostream>
#include <vector>
#ifdef USE_BOOST
  #include <boost/iterator/counting_iterator.hpp>
#else
  #include <ranges>
#endif

using idx_t = size_t;
using real_t = float;

int main()
{
  std::vector <real_t> x{0,1,2,3,4}, y{0,1,2,3,4};
  real_t a (5.);

  auto saxpy = [x = x.data (), y = y.data (), a]
               (idx_t i)
               { y[i] += a * x[i];};


```

Saxpy - fast - full example II

```
  #ifdef USE_BOOST
    boost::counting_iterator <idx_t> first(0), last(x.size ());
    std::for_each (first, last, saxpy);
  #else
    auto indices = std::views::iota (static_cast<std::vector<real_t>::size_type>
         (0), x.size ());
    std::for_each (indices.begin (), indices.end (), saxpy);
  #endif

  for (idx_t i=0; i<x.size(); ++i)
    std::cout << 5 << "x" << x[i] << "+" << x[i] << "=" << y[i] << std::endl;
  return 0;
}
```

Compile with `g++ -std=c++20 -Wall saxpy.cpp -o views` , or:

```
source /u/sw/etc/bash.bashrc
module load gcc-glibc
module load boost
g++ -std=c++20 -Wall -I$mkBoostInc -DUSE_BOOST saxpy.cpp -o boost
```

and test with `./views` or `./boost`

Notes on parallel execution - I

- you can modify the above saxpy.cpp to run in parallel, adding `#include <execution>` and `std::execution::par_unseq` in the `for_each`
- if you use g++ (or clang++), you have to link against a parallel library, such as tbb. In our case, eg with boost, besides the above steps, we have:

```
module load tbb
g++ -std=c++20 -Wall -I$mkBoostInc -I$mkTbbInc -L$mkTbbLib -ltbb -DUSE_BOOST
     saxpy.cpp -o boost+tbb
./boost+tbb
```

- if you have nvhpc sdk (software development toolkit), you can compile also with
  - cpu: `nvc++ -stdpar=multicore -std=c++20 -O4 -fast -march=native -Mllvm-fast -DNDEBUG -o saxpy saxpy.cpp`
  - NVIDIA gpu: `nvc++ -stdpar=gpu -std=c++20 -O4 -fast -march=native -Mllvm-fast -DNDEBUG -o saxpy saxpy.cpp`

Notes on parallel execution - II

- mind that GPUs are not CPUs with more threads; the hardware solutions introduced to achieve their parallelism and memory bandwidth requires constraints on the code, which currently, for nvc++, can be translated in the following recommendations:
  - prefer declaring and defining functions to be run on device in the same file (eg, define a full functor in a .hpp)
  - prefer dynamically allocated containers, such as std::vectors instead of std::arrays
  - avoid function pointers
  - avoid references in lambda captures
  - avoid exeptions

Notes on parallel execution - III

- for performance metrics, check the uploaded "bandwidth.cpp" example. You can compile with
  - `g++ -std=c++20 -Wall -Ofast -march=native -DNDEBUG -o daxpy_g++_tbb bandwidth.cpp -ltbb`
  - `clang++ -std=c++20 -Wall -O3 -ffast-math -march=native -DNDEBUG -o daxpy_clang++_tbb bandwidth.cpp -ltbb`
  - `nvc++ -stdpar=multicore -std=c++20 -Wall -O4 -fast -march=native -Mllvm-fast -DNDEBUG -o daxpy_nvc++_multicore bandwidth.cpp`
  - `nvc++ -stdpar=gpu -std=c++20 -Wall -O4 -fast -march=native -Mllvm-fast -DNDEBUG -o daxpy_nvc++_gpu bandwidth.cpp`
- and compare the results of, eg
  - `./daxpy_g++_tbb 1000000` and `./daxpy_nvc++_gpu 1000000`
  - `./daxpy_g++_tbb 100000000` and `./daxpy_nvc++_gpu 100000000`

  for `real_t=float` and `real_t=double`

Extra

You can also use the Thrust library from NVIDIA, or its AMD counterpart (rocThrust), to write computational kernels on both CPU and GPUs. In saxpy.cu you can find an example which relies on Thrust only; it can be compiled with

- `nvcc saxpy.cu -DNDEBUG -O4 -o saxpy.cpu -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_CPP`
- `nvcc saxpy.cu -arch=sm_89 -DNDEBUG -O4 -o saxpy.gpu`

to be run respectively in serial on the CPU or in parallel on a single GPU. As an exercise, you can try

- other Thrust backends
- a unified version, using std::execution, Boost and TBB on the CPU, Thrust::device on the GPU

References

- https://github.com/openpbs/openpbs
- https://en.cppreference.com/
- https://developer.nvidia.com/
- https://www.boost.org/
- https://github.com/ericniebler
- https://github.com/NVIDIA/thrust