M.Sc. in High-Performance Computing
5614. C++ Programming
Assignment 6. Concurrency

R. Morrin <rmorrin@maths.tcd.ie>

March 3, 2025

Instructions

- Gather all your code and pdf/text files into a single tar-ball.
- Provisional deadline 25th April. I might be able to extend it depending on exam timetables. I will update via email. But don't let it interfere with starting into your projects.
- Any questions, please ask me in class or email me at rmorrin@maths.tcd.ie.
- All non-code answers should be submitted in a single pdf or text file. If you write and scan in, please make sure I can read it!
- The non-code questions only need short answers - not an essay!
- Read notes at end of this document.
- Remember to use comments when appropriate.

Background

Convex Hull

A convex hull is simply the smallest convex polygon that encloses a set a points. "Convex" simply refers to the fact that a straight line drawn between any two points inside the polygon lies entirely inside the polygon. They

One way to think of the convex Hull is that if you extend a line through any adjacent points on the hull, than all other points must be on one side only of that line. See Fig. 1. We will use this fact to determine the hull.

Orientation of points

Given two vectors $\vec{v}_1 = x_1\hat{i} + y_1\hat{j} + z_1\hat{k}$ and $\vec{v}_2 = x_2\hat{i} + y_2\hat{j} + z_2\hat{k}$. You can probably recall from your first year classes that the cross product of two vectors $\vec{v}_1 \times \vec{v}_2$ is orthogonal to both $\vec{v}_1$ and $\vec{v}_2$ and its direction is given by the "right hand rule". If we have two vectors in the ij-plane so that $z_1 = z_2 = 0$ then the resulting $\vec{v}_1 \times \vec{v}_2$ will be given by $\alpha\hat{k}$ and the sign of α will depend on the orientation of the vector. The direction of the cross

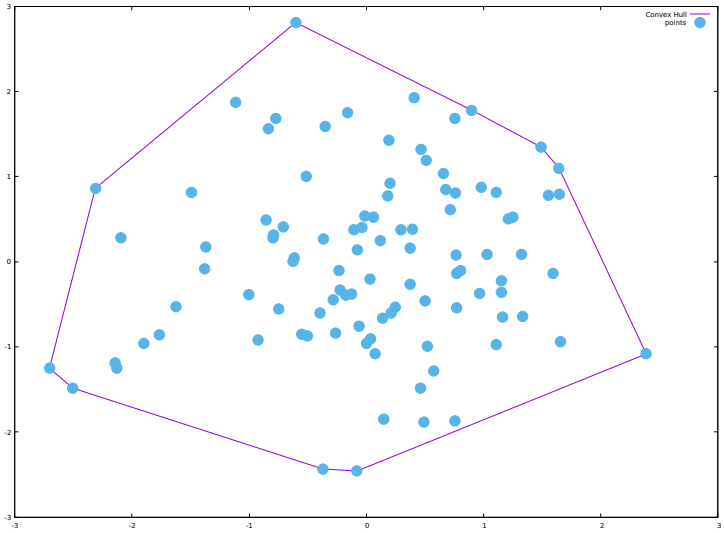

Figure 1: Convex hull around a set of 100 random points.

product result depends on whether the points were taken in a clockwise or anti-clockwise order.

We can use this for the convex hull. Suppose we have a point p_1. Then the cross product of $\vec{p}_1$ with vectors through any points on one side of the half plane it defines (i.e. the line through p_1 to the origin) will have the same orientation (i.e. sign in the $\hat{k}$ direction). Whereas points on the other side of the line will result in a cross product which has opposite sign. (This is just the right hand rule again). See Fig. 2.

Putting all that together, if point $\vec{h}_1 = (x_1, y_1)$ is on the convex hull, then some other point $\vec{h}_2 = (x_2, y_2)$ is the next point along the hull if and only if for all other $\vec{h}_i = (x_i, y_i)$ we have that[1] $(\vec{h_2} - \vec{h_1}) \times (\vec{h_i} - \vec{h_1})$ has the same sign $\forall i$. You can use the mnemonic for cross product in terms of determinants as

$$\begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ x_2 - x_1 & y_2 - y_1 & 0 \\ x_i - x_1 & y_i - y_1 & 0 \end{vmatrix} = \begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_i - x_1 & y_i - y_1 \end{vmatrix} \hat{k} \tag{1}$$

and the sign of the determinant will tell us whether the points are ordered clockwise or anti-clockwise.

Merging two hulls

See Fig. 3. Suppose that you have two convex hulls and you want to merge them so that there is one overall hull enclosing all the points. There are different ways to do it, but I think the below is easiest to follow. You can use another method if you wish. Although don't just create a set of points from the three convex hulls and then solve for the hull on that set. If you are stuck, you can do that, but I won't be able to give full marks for that way.

[1] Subtracting $\vec{h}_1$ is only translating the three points so that $\vec{h}_1 \to (0, 0)$ and the points look as in Fig. 2

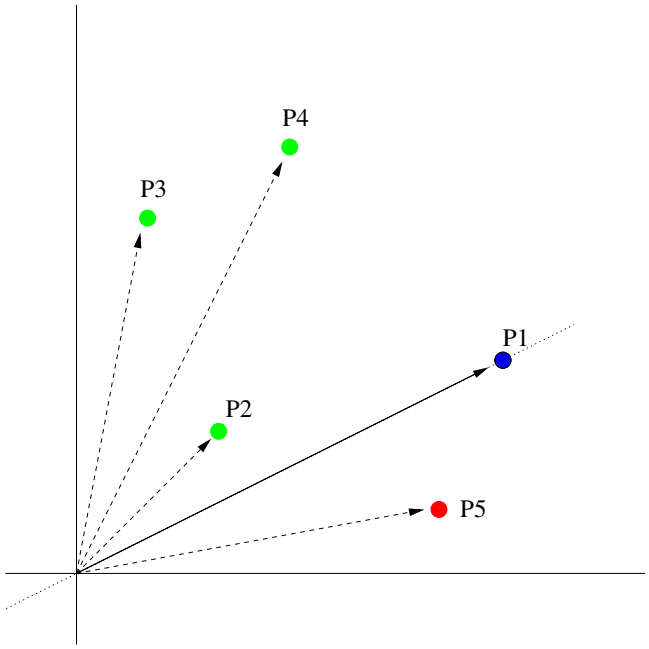

Figure 2: The right hand rule tells us that the cross product of $\vec{p}_1$ with $\vec{p}_2, \vec{p}_3, \vec{p}_4$ will be in the same direction whereas the cross product of $\vec{p}_1$ with $\vec{p}_2$ will be in the opposite direction and hence opposite sign.

1. Find rightmost point (A) on left upper hull and leftmost point (B) on right upper hull
2. Define a vertical line at arbitrary x_m between then. You can use midpoint. Keep the same vertical line for remainder of algorithm.
3. Draw a line connecting (A) and (E). Note the value of y_m which is the $y-$coordinate where it intersects the line $x = x_m$.
4. Now rotate clockwise around the right upper hull to the next point (F). Try a line between (A) and (F). If the intersection of that line with the vertical line occurs at a point $y_i > y_m$ then set new $y_m = y_i$ and keep $\vec{\text{AF}}$ as possible solution.
5. Continue to rotate to (G). Try the same as above with $\vec{\text{AG}}$. This intersection is lower so reject $\vec{\text{AG}}$ and go back to $\vec{\text{AF}}$.
6. Now repeat for left upper hull by rotating anti-clockwise. We start with $\vec{\text{FA}}$ so we rotate from (A) to (B). Solve for the intersection $\vec{\text{FB}}$ with the vertical line. This is better so this gives the new y_m and keep $\vec{\text{FB}}$ as new possible solution (forgetting previous candidate).
7. Continue and try $\vec{\text{FC}}$. Accept this.
8. Continue and try $\vec{\text{FD}}$. Reject.
9. Now back to right hull and try $\vec{\text{CG}}$. Accept.
10. Continue and try $\vec{\text{CH}}$. Reject.
11. Now back to left hull and try $\vec{\text{GD}}$. Reject.

As you can see, the simple algorithm continues switching between sides after a rejection, until you get two rejections in a row. The one that you

are left with will be the segment for the larger merged hull. The new hull would have D $\rightarrow$ C $\rightarrow$ G $\rightarrow$ H.

You should be able to code this up using, for example, a double (i.e. nested) while loop. The outer loop could rotate around the points on the left, whereas the inner loop could rotate around the points on the right. The procedure is finished when you get two rejections in a row.

The analagous procedure for joining the lower hulls should be obvious once you can do the upper.

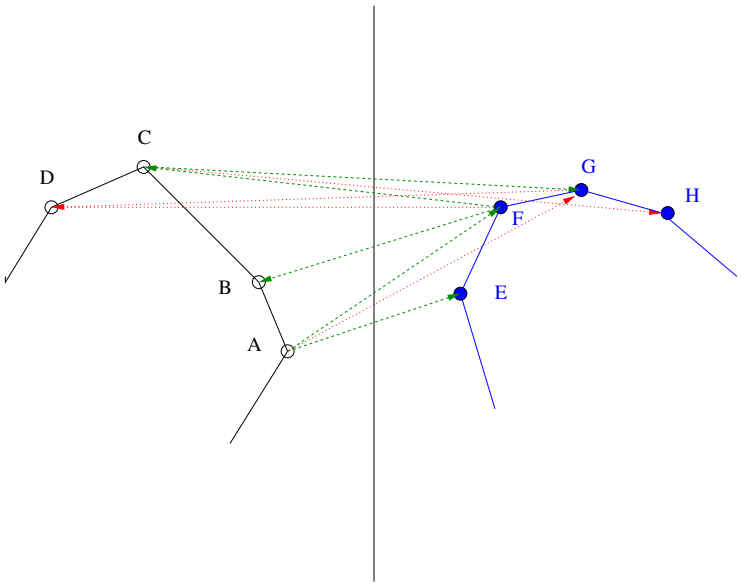

Figure 3: Merging two hulls. Method for upper part shown

Note on member functions and the `this` pointer.

We did use the this pointer earlier in the lectures but I did not go into it in detail. For the purposes of the second part of this assignment you will need to know a little more. A decent short article is here `https://www.learncpp.com/cpp-tutorial/8-8-the-hidden-this-pointer/`.

Basically, when dealing with member functions, the way that the program keeps track of what object the function is being called on, is to pass the this pointer as a hidden first argument. For example, if you have

```
class myclass
{
public:
    myclass () : x{0} {};
    void do_work(int n) {x +=n;};

private;
    int x;
};
```

and in the code you do

```
myclass A;
A.do_work(10);
```

then the compiler actually processes the code as if it was

```
void do_work(myclass * this, int n){ //as if non-member function
    this->x+=n;
}
```

called via

```
myclass A;
do_work(&A,10);
```

The relevance of this to part 2 of the assignment is that you will be passing a member function to std::async. To launch the `do_work` function in a task for a particular instance of `myclass` you would need to explicitly pass this normally hidden parameter as the compiler cannot generate it automatically. In other words, the first argument after passing the member function should be the address of the object for which you want to call that function. If we were to do it for the example above, we would need

```
myclass A;
auto fut=std::async(std::launch::async, &myclass::do_work, &A, 10);
```

Note on passing references to `std::bind` & `std::thread`

I mentioned this very briefly in one of the earlier lectures where I used `std::ref`, but I did not elaborate. The arguments to `std::bind` are copied or moved, and references are not forwarded to the underlying function unless wrapped in std::ref (or std::cref for const object) which would wrap the object up in a std::reference_wrapper. The same thing is true for a std::thread. If an argument needs to be passed through to a thread function by reference then you need to wrap it.

For example, suppose that you have a function which takes a reference to an int as a parameter

```
void f (int &r);
```

Then if you want to create a thread which runs that function you will need

```
int result {0};
// std::thread {f, result};          // Will not work
std::thread {f, std::ref(result)};
```

The provided code passes the random number engine to `std::bind` wrapped in a std::ref. If you do not do this, then `std::bind` causes a regular reference to decay to pass-by-value which means that you pass through the state of the engine to the underlying function "by value". When you use it inside that function, the rng state is modified within that function for each random number pulled from it but the state is not updated back in the calling function. The impact of this is that if you bind the same rng into two functions, each function will start with the same state and pull the same sequence of numbers. In order for the "global" state to be updated, you need to preserve the reference by wrapping with a std::ref when passing through `std::bind`, even if you would not need to do it if calling the function without `std::bind` i.e. if `f(rng)` would work fine where `f` expects you to pass `rng` by reference, you would need to do `std::bind(f, std::ref(rng))` when using `bind`.

Q1. Parallelising `std::inner_product` (25%)

We introduced the new Execution Policy which is available since C++17 for many (but not all) of the Standard Library algorithms out of the box. For this part of the assignment you will "parallelise" the computation of a `std::inner_product`.

```
template< class InputIt1, class InputIt2, class T >
T inner_product( InputIt1 first1, InputIt1 last1,
                 InputIt2 first2, T init );
```

You will just consider the usual definition[2] $r = \vec{a} \cdot \vec{b}$ or $r = \sum_{i=0}^{n-1} a_i b_i$

The `main` function for Q1 is included in assignment6a.cc. You will write the rest of the code within this `main` function.

The code defines two std::vector<double>s and populates them with random numbers from the standard normal distribution. You will calculate the dot product of these two vectors. The serial version or the dot product and timing code is provided. You will need to parallelise using std::thread with a std:packaged_task and also with std::async.

A suggested useful item for this section would be to define a generic named lambda

```
auto partial_dot = [](auto it, auto it2, auto it3){
    return std::inner_product(it, it2, it3, 0.0);
};
```

which just acts as a wrapper for the standard library `std::inner_product` function. This is included in the file for you. You can do it a different way if you want.

A) `Makefile` (5)

As per usual, write a `Makefile` to manage the assignment. I should be able to run `make all`, or just `make` to compile all needed targets or I should

[2]There is another more general version of the function in the standard library that takes two binary operations as arguments.

be able to run any of `make assignment6a`, `make assignment6b` or `make assignment6c` individually. Include also a target for `make clean`. The same `Makefile` is for all the rest of the assignment. You will need `mpicxx` for `assignment6c` and also link that to `libboost_mpi` and `libboost_serialization` libraries.

B) `std::packaged_task` and `std::thread` (10)

Parallelise the computation of the dot product across a number of threads. The purpose of the assignment is to just practise concurrency techniques so you can just pick a fixed number of threads such as 3. (I use 3 as an example as the number of available hardware threads on my laptop is 4. The number of hardware threads itself does not place a hard limit on the number of software threads; if there was more they'd just get swapped in and out). If you want, you can do something a little fancier in terms of choosing the number of threads dynamically, but it is not required or expected.

Note that if you wanted to create a std::packaged_task for a function `int myfunc(std::vector<int>::iterator a);` that takes an iterator (over a std::vector<int>) as a parameter and returns an int then you would have to define something like

```
std::packaged_task<int(std::vector<int>::iterator)> pt1 {myfunc};
```

Two more points on std::packaged_tasks.

1. They cannot be copied so will need to be "moved" into a std::thread. E.g. `std::thread t1 {std::move(pt1), arg1};`
2. **Before** they are "moved", you will need to store the std::future via something like `auto f1 = pt1.get_future();`

C) `std::async` (10)

Repeat the above parallelisation using `std::async`. Make sure that the answers agree. Note that due to the way that floating point numbers are stored, in theory you might not get exactly equal answers when the orders of summing up are changed due to machine precision and roundoff. Note that in the provided file I defined a global variable

```
const auto policy = std::launch::async;
```

and you can use this to make it easier to switch between launch poicies for all your tasks with one line. If you were using `std::async` in your own projects, this might be helpful for debugging issues if you could switch all tasks to deferred.

You can time the execution of your code and compare it to the serial. Check the run times using `std::launch::async` and `std::launch::deferred` policies for `std::async`. For Q4 you will provide a very short description of what you see in a text file or pdf.

Sample output with `std::launch::async`

```
Num hardware threads = 4
Dot product (serial). Answer = -5067.91
Elapsed time : 744 ms

Dot product parallel async: dot_prod = -5067.91
Elapsed time : 475 ms

Dot Product parallel threads & packaged task  = -5067.91
Elapsed time : 426 ms
```

Sample output with `std::launch::deferred`

```
Num hardware threads = 4
Dot product (serial). Answer = -3321.88
Elapsed time : 716 ms

Dot product parallel async: dot_prod = -3321.88
Elapsed time : 712 ms

Dot Product parallel threads & packaged task  = -3321.88
Elapsed time : 390 ms
```

Q2. Convex Hull (45%)

A) Point `class` (20)

Download the file point.h. This file contains a class definition for `Point` which is very similar to Assignment 4. Included is an overload of the output stream operator `operator<<` and a simple constructor `Point(double inx, double iny)` as well as an explict default contructor declaration. You do not need to write these. You can modify the output stream operator if you want the output in a different format. It is currently comma separated.

You will need to write four function defintions in point.cc

(i) Equality Operator

Write the function defintion for the member function

```
bool Point::operator==(const Point & rhs)
```

which simply returns true if and only if the values of both `Point` member variables x and y on the left hand side are equal to the corresponding values of the `Point` on the right hand side of the operator. Else return false.

(ii) Cross Product

Write a function which returns true if the cross product of two vectors is orientated in the negative $\hat{k}$ direction. In other words, calculate $\vec{v}_1 \times \vec{v}_2$ where $\vec{v}_1$ and $\vec{v}_2$ and return true if the the result is negative (or zero, for collinear points). The vectors are passed in as three `Point`s. $\vec{v}_1$ is the vector between `p1` and `p2` and $\vec{v}_2$ is the vector between `p1` and `p3`. See

Eq. 1. (But check the calculation for yourself in case I made any typos in the document!).

```
bool cross_prod(Point p1, Point p2, Point p3);
```

(iii) Sort `std::vector` of `Points`

Write a function

```
void sort_points(std::vector<Point>& points)
```

which sorts the vector of points in ascending order of $x-$coordinate. If two points have the same $x-$coordinate, then sort between those on their $y-$coordinate. Use the `std::sort` function from `<algorithm>`. Write your own comparison function. I would suggest using a named lambda expression.

(iv) Writing to file

Write a function

```
void write_to_file(std::string fn, std::vector<Point> pts)
```

which writes the std::vector of points to the file named `fn`.

B) ConvexHull `class` (20)

Download the file ConvexHull.h. This contains the class definition for `ConvexHull`. You should write the below functions in ConvexHull.cc.

(i) Constructor

Write a constructor for this class

```
ConvexHull::ConvexHull(std::vector<Point>::iterator beg,\
                       std::vector<Point>::iterator end)
```

which takes two iterators as parameters. The iterators should be for a std::vector<Point> which has been sorted using the `sort_points` function which you wrote earlier. The `Points` between these iterators [beg, end) are added to the `std::vector<Point> points` member variable.

(ii) Finding the Hull

Write the function definition for

```
std::vector<Point> generate_hull()
```

which calculates the points on the convex hull surrounding the points stored in the `std::vector<Point> points` member variable. The resulting points should be stored in the `std::vector<Point> hull` member variable.

Because you (should) have already sorted the `Point`s then the first element in the vector should be the `Point` with the leftmost $x-$coordinate. This point (p_1) is necessarily on the convex hull. To find the next point on the hull, simply find the point p_2 such that all other points p_i are on one side of the half plane defined by the line through p_1 and p_2. Use the `cross_prod` function which you wrote above. (Note that the choice of returning true or false, or equivalently $>$ or $<$ in the return statement will just determine whether you find the hull in the clockwise or anti-clockwise direction).

When you get to the end of the std::vector<Point> you will only have the bottom part of the hull and you will need to traverse the vector in reverse to add the points to the top of the hull.

This function should be 30-40 lines.

(iii) Merging the hulls

When we have two adjacent hulls we will merge them to find an overall overlapping hull. One method to do this is decribed above and shown in Fig. 3.

You will write the function defintion for

```
void ConvexHull::merge_to_right(const ConvexHull &right)
```

as a quick-and-dirty way to do this. If you do this the "long" way, with repeating code, then this function will get quite long - perhaps over 150 lines. However, it is ok for this function, however you are welcome to tidy it up if you wish. Write it so that if you call

```
A.merge_to_right(B);
```

then after the function completes, `A` should have "stolen" all the points from `B` so that its member variables `points` & `hull` hold all the individual points and the points on the overall hull. `B` will now be empty. The below would be one suggestion:

1. Split both hulls into upper and lower hulls. The extreme $x-$coordinate `Point`s in each hull will be in both upper and lower hulls. So you might store the top of the hull to the left in `std::vector<Point> upper_left_hull` for example.
2. I suggest storing the points in the order that you will use them so that for Fig. 3 `upper_left_hull` would contain $\{A, B, C, D \ldots\}$ and `upper_right_hull` would contain $\{E, F, G, G \ldots\}$ in that order.
3. Do the algorithm to find $\vec{CG}$ as the maximum.
4. Repeat the above steps for `lower_left_hull` and `lower_right_hull`. You can just do a naive cut-and-paste with a slight edit for this one[3].

[3]If you want, you can put the logic into a different function and pass in a parameter to

5. Now that you have the joining segments, you need to finally stitch everything into the overall outer hull.

C) Putting it all together (5)

Download the file assignment6b.cc which has the bones of the `main` function. You actually only need to add a few lines to this file. The code for calling the serial calculation is included. The locations are between the C-style `/* */` braces. You will need to

1. Construct three `ConvexHull` objects, each from one-third of the overall points which are generated for you earlier in teh file to construct the `CH` object for the serial calculation.
2. Call the `generate_hull` method on each of these. Run the three as asynchronous tasks using `std::async`.
3. After all three tasks are spawned, you can call `wait` on them. You would need this in the case of running with `deferred` launch policy.
4. The rest of the code, such as calling the merging functions is provided.

Output

```
> ./assignment6b
Serial time: 23817 ms

Parallelised time 12022 ms
```

1 Q3 Boost.MPI (20%)

For this part, you will modify `assignment6b` to parallelise the program using `Boost.MPI` rather than threads. You will need to install `Boost.MPI` and `Boost.Serialization` on your own machine or else load up the correct module for `chuck` to set up your environment (See L20 slides).

You can keep the strategy the same as for the threads - i.e. generate and divide the points into three subsets by processor 0. Then send one-third of the vector of `Points` to each of processor 1 and 2. Each processor will compute the hull for its points (You have this written already) and then processors 1 and 2 send their computed hulls back (again as a vector of `Points`) to processor 0 which does the merge.

The files you will need to modify are:

a) `point.h` and `point.cc`

You will need to add a templated function

```
template <typename Archive>
        void serialize(Archive & ar, unsigned /*ver*/);
```

indicate upper or lower and use that to make the changes to the algorithm decision, so the same code can process upper and lower joins, but you don't need to. Just get it working for this

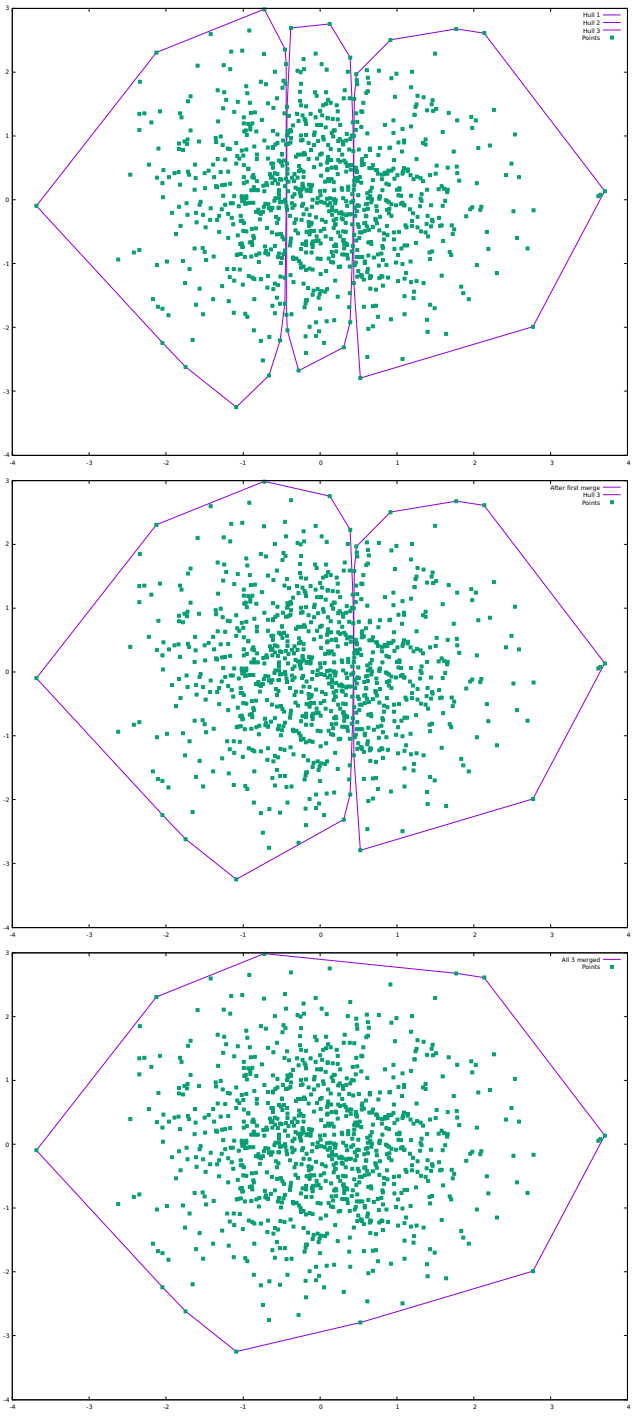

Figure 4: These plots show the different steps in the merging. The top plot shows three individual hulls. The next one shows the state after the first two have been merged and the final one shows the state after the third one has been merged.

which will serialise the `Point` class. (If you want to do the templated function the C++20 way, then you can do it as `void serialize(auto & ar, unsigned /*ver*/);` and use flag `-fconcepts` or `-fconcepts-ts` with an older compiler on C++17 standards). You will need to explicitly instantiate versions of this function for both `boost::mpi::packed_iarchive` and `boost::mpi::packed_oarchive` template parameters. You will also need to include <`boost/mpi.hpp`>.

b) `Convexhull.h`

The only thing that I suggest adding to the `ConvexHull` class is to add a very short public function that will move a vector of `Point`s into the member variable `hull` of that class.

```
void movehull(std::vector<Point>&& in)
```

c) `assignment6c`

You will need to create assignment6c (Use assignment6b as a template ... copy it and take out the `std::thread` and `std::async` bits etc.).

Suggested approach:

- Generate the `Point`s on processor 0.
- Send one-third of these to each of processor 1 and processor 2. Keep one-third for processor 0

  (Remember that your constructor for `ConvexHull` takes iterators to the start, and end, or a sequence of elements in a `vector` of `Point`s)
- Call `generate_hull()` member function for each processor
- Send the generated hulls from processors 1 and 2 back to processor 0. (as `std::vector<Point>`s).
- `std::move` these returned `vector`s into `ConvexHull` objects on processor 0
- Merge the hulls the same as `assignment6b`.

Q4. Written Questions (10%)

You can check the timings for Q3 if you want, but I would not expect to see a good performance for using `MPI` for this setup.

A) Timings. Q1 & Q2 (5)

(i) For Q1

For Q1, just give a very short description of your timing results for running your code. You can also include output for running with both `async` and `deferred` policies. What is the meaning of these different policies and what is the default policy if you do not specify one?

(i) For Q2

Provide some short timing output for running Q2, with `-O0` and `-O3` levels of optimisation.

Also run Q2 with `deferred` launch policy. Do you see any difference compared to the serial timings? Does it tend to be faster or slower or roughly the same?

B) Plots for Q2 (5)

Just include three of your result plots the equivalent of Fig. 4. They output files of points should be generated automatically for you if you use the provided `main` function and everything works.

Notes:

- For both `main` functions, I just hardcoded a size `N` for the number of `Point`s etc. Modify this to suit your own hardware! If it is too small you might not see any noticeable difference in the timings but if it is too large, it might cause a seg fault depending on the memory available in your machine.
- You don't need to do exactly as I suggest, once the overall idea is followed and you can divide the problem into say 3 hulls and then merge the results. You can use different algorthms to find the hulls or do the merging. Just leave comments if you use an alternative etc.
- Prefer to use smart pointers over raw pointers in general.
- Put some sanity checks into your code. Regular checks are fine - you don't need to worry about exceptions or asserts etc. Just some simple logic checks such as bounds checking etc. However, if you encounter an error for this assignment, throw an error message rather than an exit.
- Use member list initialisation where possible.
- Use universal initialisation syntax {} where you can rather than = or () notations.
- 10% of the respective marks available for each of the coding questions will be given for comments. You do not need to go overboard and please make sure the comments are relevant. i.e. you don't need to say obvious things like `int A; /*This is an integer */`. Note that in the lecture slides, I sometimes include comments such as `/* Function Prototype */` - but this is just for the purposes of the lectures. This would not necessarily be a helpful comment on its own in a proper program. Strive to make your code as self-documenting as possible by using appropriate names for variables and functions.
- To create your tarball, `cp` everything you want to submit into a directory e.g. `Assignment6`, and then change directory so that you are in the parent directory of `Assignment6`.

  `tar -cvf [username].assignment6.tar Assignment6`

  where you replace `[username]` with your own username.