Getting Started

GitHub: https://github.coventry.ac.uk/ab0475/6004CEM_OPEN/tree/master/20-21

Cisco VPN :
https://anyconnect.coventry.ac.uk/+CSCOE+/logon.html?reason=12&gmsg=504241415250472D454E2D494341#form_title_text

VPN connection url: https://anyconnect.coventry.ac.uk

Username: akhnoukp
Password: Ze-4kPgY

MobaXterm: https://mobaxterm.mobatek.net/

SSH: ssh -Y stamp10@hamarabi
Password: akhnoukh17

Submission content requirements for the entire portfolio

All elements, including required input and output files (as specified in each portfolio element), should be placed in folders indicating which aspect of the portfolio they relate to, then these should be placed in a folder named portfolio which is then compressed to create a single file that you will then submit.

Parallel Programming Portfolio Element

Preamble

To have the best chance of a good grade in this element you are strongly advised to complete the exercises for Parallelism. Every aspect must be written in C++.

Part A

Write a program capable of accepting one of the available schedule types at runtime. It doesn't matter whether the program itself is going to be able to take **full** advantage of this variable schedule, only that it can be set, and **at least one parallel regions** in your program makes use of it.

Part B

The following problem may seem too trivial to need OpenMP, and in the real world it would be. But this isn't the real world, it's your assignment, and not using OpenMP will cost you a great many marks because you will not have solved the correct problem.

Using what you have and these ten vectors, create a parallel application that solves the following problem:

5 14 10

7 -8 -14

-2 9 8

15 -6 3

12 4 -5

4 20 17

-16 5 -1

-11 3 16

3 10 -19

-16 7 4

Assume each of the ten three number lines is a position in a Cartesian co-ordinate system.

Further assume each represents an object moving by one step in a random direction in one of the three available directions per iteration.

Set your **particle position alteration variable** to 2 to start with

Two of the three values of each particle (as represented by every set of three values) can be altered per step, being either incremented or decremented using your **particle position alteration variable**. It doesn't matter if they go into negative values (you'll note many of the ones you've been given start with at least one), since these are moving in three-dimensional space.

Increase the **particle position alteration variable** by any integer value from one to five on each iteration.

Now using your practice building a basic time series (I assume here you've had a go at the preparation exercises), iterate these vectors through fifty steps.

Your program should print the original state of these vectors, their state at twenty-five steps, then their state at the fifty-step point.

Part C

Using the code and results from before as a starting point.

Locate the centre for this set of particles (the point around which they are all grouped).

For another fifty steps, move the vectors towards this point by using the same method as before, after setting the **particle position alteration variable** back to 2.

Again, your program should print the original state of these vectors, their state at twenty-five steps, then their state at the fifty-step point.

Print both the final distance between all the particles and this central point and their original distance from it.

Move the particles back towards this central point you've identified using the **particle position alteration variable.**

Quick tip – comments in your code are a good idea here.

Marking Rubric

	Program functionality		Adherence to assignment specification		Code Quality	
First **≥70**	Code exceeds expectations and always runs successfully.		Meets and may exceed all assignment specifications, design matches requirements. Uses OpenMP.		Code quality is excellent, has comments and proper layout. All indenting is correct.	**First** **≥70**
Upper Second **60-69**	Code is complete and always runs successfully,		Meets most assignment specifications, design matches requirements. Uses OpenMP.		Code quality is good, has comments and proper layout. Still some indenting errors.	**Upper Second** **60-69**
Lower Second **50-59**	Code is complete and compiles. May not always run successfully. May not be in C++.		Meets more assignment specifications, design still not what is required, may not use OpenMP.		Code quality is fair, may still lack comments or have improper layout. Indenting should be correct at this level.	**Lower Second** **50-59**
Third **40-49**	May be complete but might not compile. May not be in C++.		Meet some assignment specifications but doesn't do so in a satisfactory way.		Code quality is poor, lacks comments, improper layout.	**Third** **40-49**
Fail **<40**	Code is incomplete. May not be in C++.		Fails to meet any assignment specifications.		Quality is poor, or there isn't enough to judge this metric.	**Fail** **<40**
Late	0		0		0	**Late**

Distributed Programming Portfolio Element

Part A

Write a program which collects data on the total data processing capabilities of the cluster, displayed on a node by node basis.

- This should include:
- Node Count (head node only)
- Then, by Node name (which should be clearly shown):
- Number and clock speed of cores, real or virtual.
- System RAM of the current node.

All these outputs may be printed to screen by each core on the nodes about their own stats. Depending on how your code is written they could be printed to screen as disjointed lines of texts, thus making no sense. Try to avoid this using the techniques you've been taught.

Part B

Write a program using that loads a poem **A Woman's Heart by Marietta Holley** into a vector. You can find it here Git Repository.

Using a combination of OpenMP and MPI do the following:

For the following you should **only** be using a **CUSTOM MPI DATA STRUCT** to transfer information between nodes. Failing to do so will impact your grade.

Split this poem up a line at a time, sending entire lines from the head node to randomly selected cluster nodes. Keep the original poem on the head node.

These nodes should store the lines they've been sent, and be able to return a count of the lines they've been given, on being asked to do so by the head node, in the following form

[[total lines held] [original index of line 1] [original index of line 2] [original index of line 3]]

And so on, for however many lines the node has been sent, so it might be **[[3][17][6][9]].** Being only integers (the brackets wouldn't be included), it shouldn't be hard to create the corresponding MPI data structure.

The Poem itself can be sent in its entirety, but the nodes can only own the lines they've been given the indexes of.

It is the case that with MPI all nodes already have access to the entire poem, but this assignment is about writing code for distribution, so sending the poem in code again to each node will increase your grade.

Part C

Using the collected indexes, build a new copy of the poem by asking the nodes with the lines you need to send them back to the head node as you need them, while recreating the poem. Then display the finished/rebuilt poem, only once **on the head node only** on completion of this process.

Marking Rubric

	Program functionality	**Adherence to assignment specification**	**Code Quality**	
First ≥70	Code exceeds expectations and always runs successfully. Distribution aspects excellent.	Meets and may exceed all assignment specifications, design matches requirements.	Code quality is excellent, has comments and proper layout. All indenting is correct.	**First ≥70**
Upper Second 60-69	Code is satisfactory Distribution aspects fair, may not have all requirements fully implemented, but those that are implemented work.	Meets most assignment specifications, design matches requirements. MPI used correctly, but incompletely in terms of the brief.	Code quality is good has comments and proper layout. Still some indenting errors.	**Upper Second 60-69**
Lower Second 50-59	Code is complete and compiles. May not always run successfully. Distribution elements poor. May not be in C++.	Meets more assignment specifications, design still not what is required. MPI used, but incorrectly.	Code quality is fair, may still lack comments or have improper layout. Indenting should be correct at this level.	**Lower Second 50-59**
Third 40-49	May be complete but might not compile. May not be in C++.	Meet some assignment specifications but doesn't do so in a satisfactory way. MPI not used.	Code quality is poor, lacks comments, improper layout.	**Third 40-49**
Fail <40	Code is incomplete. May not be in C++.	Fails to meet any assignment specifications.	Quality is poor, or there isn't enough to judge this metric.	**Fail <40**
Late	0	0	0	**Late**