1 Task 1

1.1 Uploading the files

I used WinSCP to upload the files to the HPC Wales Cluster. I logged onto HPC Wales Cluster by inputting login.hpcwales.co.uk as the host, and my username michael.smith as well as my password. This allows me to log in to the server.

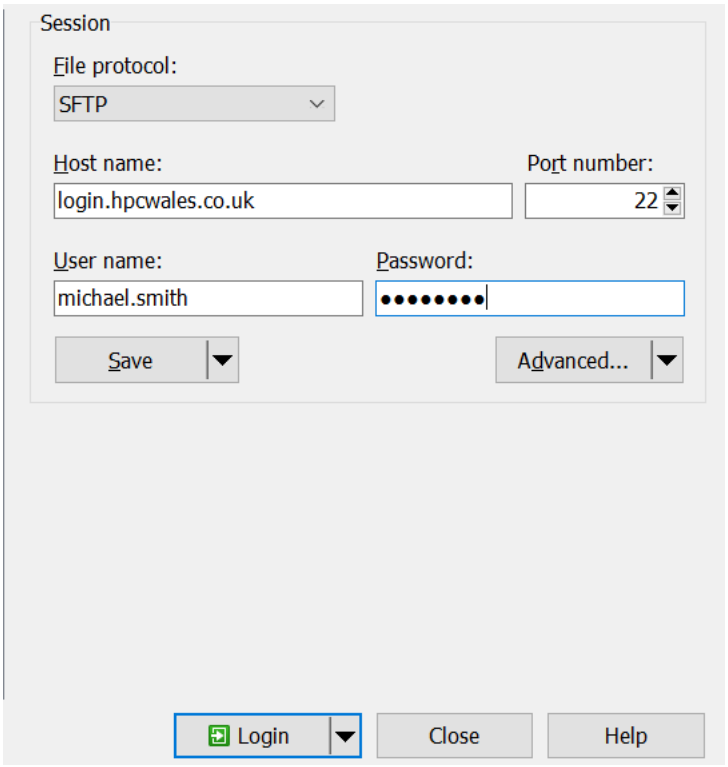

Figure 1: Logging into HPC using WinSCP

Once logged in using WinSCP I then just simply dragged and dropped the sample files from my computer, ontvo the HPC Wales Cluster.

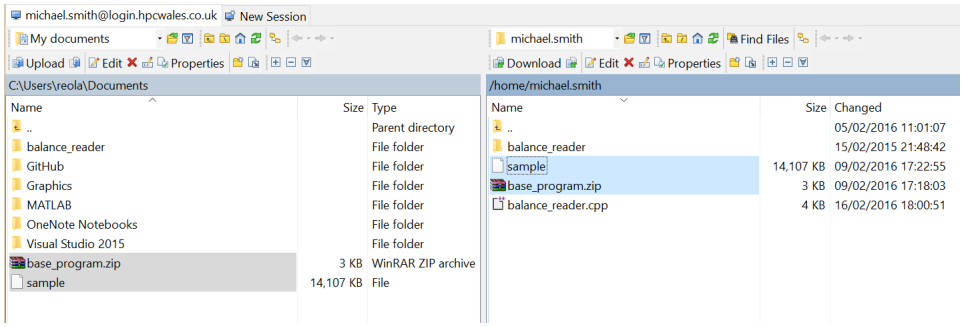

Figure 2: Transferring the files to HPC using WinSCP

When the files were transferred onto the HPC Wales Cluster using WinSCP they were originally sitting in the root Cardiff node, but I want the files to be moved to the Bangor node. To do this I need to use Putty.

To log into the HPC Wales Cluster using Putty, I used the same credentials as I used to log in using WinSCP.

To transfer the files to the Bangor cluster I need to find the host-name of the server to send them to. To do that I type hpcwhosts in the console. This brings up a table of all the clusters, including information such as their location, phase and host-name.

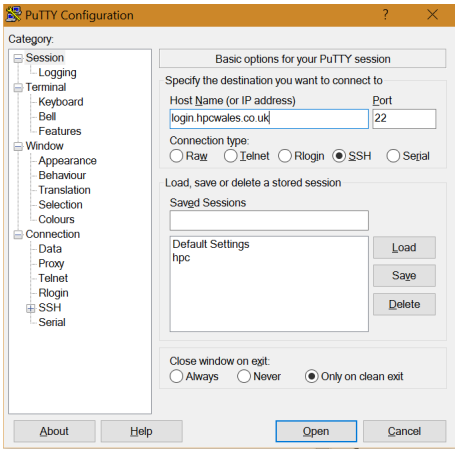

Figure 3: Logging into HPC using Putty

Now knowing the host-name I need, which is bwl001 I simply just have to scp (securely copy) the files. I do this by typing the following two commands: scp base program.zip bwl001: for the zip file, and scp sample bwl001: for the sample file.

Then to check that the files were transferred I have to log in to the Bangor cluster and check the directories there. To log in to the Bangor cluster I type ssh bwl001, just like the host name I used as the destination to transfer the files. After being asked to re-enter my password I get a message confirming that I have successfully logged into the Bangor cluster.

If I type ls into the console I can see all of the files which are uploaded to the Bangor cluster. Both the files I uploaded, then transferred across are visible in this list, base program.zip and sample.

1.2 Running the program

Once uploaded I then have to unzip the base program.zip file, to do this I type unzip base program.zip, this creates a directory called balance reader and inside this directory creates a balance reader.cpp, balance reader.o and a Makefile to compile the program. If I cd balance reader then ls I can see that this directory and the files were created successfully.

Inside the directory I type, make to trigger the makefile to compile the program. This creates the executable balance reader inside the directory.

To run the file, if I type cd .. to return back to the root directory. I then type ./balance reader/balance reader sample, the first part accesses the balance reader executable created using the make command inside the /balance reader directory, the second part is the sample file which is read in as an argument. When I run the program, it returns the running total from the sample file in the console.

2 Task 2

2.1 File Analysis

Firstly I examined what is going on in the sample file, and how the current balance reader is processing it. The current program takes every line of the sample file, one line at a time, and processes the line and adds it to a total. This is done in a single thread, which basically means one pair of eyes reading the file from the start to the end.

Since the transactions are commutative, it actually does not matter in what order I read the lines, for example I do not need to read line 34 before line 60, as long as both lines are read and processed at some point the answer is the same. Depositing 3560.31 then withdrawing 578.40 is the same as first Withdrawing 578.40 then depositing 3,560.31.

2.2 Method

Knowing this I can create more threads (more ”eyes” looking at the file) and each thread can read different parts of the file, this is all done at the same time, for example thread one will be reading line one the same time as thread two is reading line 125,001.

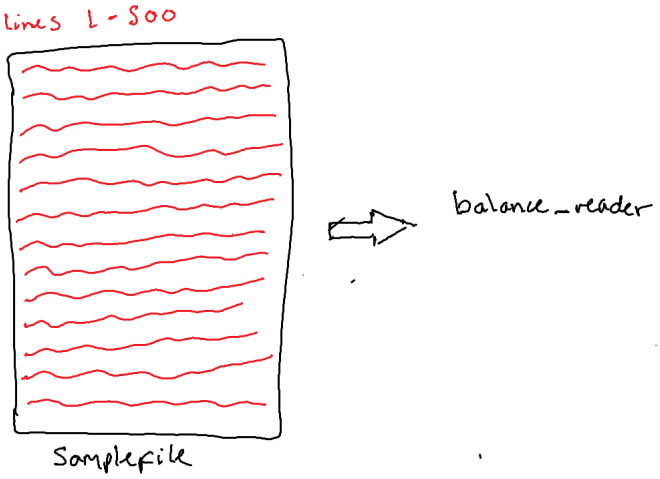

Figure 4: How the file is currently processed, red represents the single thread

I can split the file between the threads by simply dividing the amount of lines in the file by the amount threads, for example if I had four threads and 500,000 lines then each thread will have to parse 125,000 lines each (500,000 divided by four).

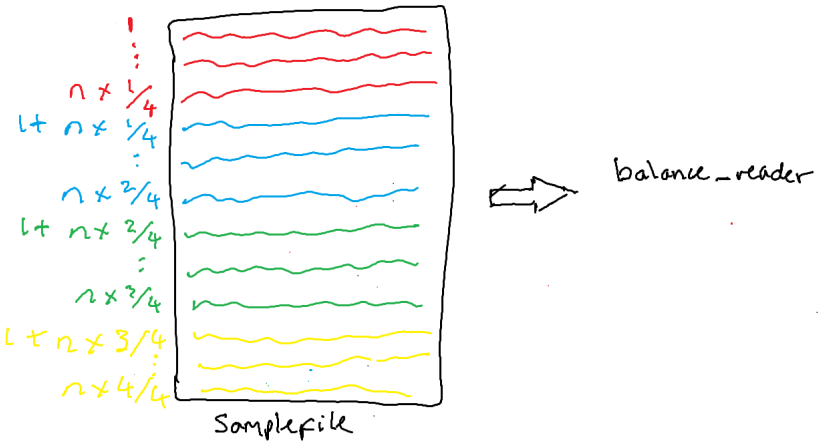

Figure 5: Proposed way of processing the file, each colour represents a different thread

Each thread will be responsible for the total of their section. For example thread one keeps a running total between lines one and 125,000. At the end of each thread, these four totals are added up to give a grand total. Having each total independent and adding up the total once all threads have finished will ensure there will not be a race condition, where one thread might overwrite the total of another.

In addition each thread will have it's own ifstream, again to combat against a race condition; if multiple threads were all accessing the same ifstream, then they will be essentially fighting over what line to process, thus potentially losing track of what line each thread is supposed to be on.

2.3 Practical

//

```
// Name        : balance_reader.cpp
// Author      : Gray, Cameron C.
// Version     : 1.0
// Copyright   : ICP-3029, School of Computer Science, Bangor University
// Description : Base Program - Read a Balance File (CSV Format) and display
//             : the final balance at the end of processing.
//============================================================================

#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>
#include <iomanip>
#include <cstdlib>
#include <pthread.h>
using namespace std;

string filename;
unsigned line_count;

/*
 * Split a std::string based on a supplied delimeter.
 * s - the string to split.
 * c - the delimeter character.
 * v - the std::vector to place the elements in.
 *
 */
void split(const string& s, char c, vector<string>& v) {
        string::size_type i = 0;
        string::size_type j = s.find(c);

        while (j != string::npos) {
                v.push_back(s.substr(i, j - i));
                i = ++j;
                j = s.find(c, j);

                if (j == string::npos)
                        v.push_back(s.substr(i, s.length()));
        }
}

void *ReadLines(void *threadid)
{
        long tid; // the thread id
        tid = (long) threadid;
        ifstream myfile(filename.c_str()); // ifstream local to this thread
        string line;
        double runningTotal = 0.0;
        unsigned section = line_count / 4; // divide the data into equal parts

        if (myfile.is_open()) {
                for (unsigned int i (0); i < section * tid; i++) // section size times t
                { // meaning it skips 0, 1 or 2 sections depending on what thread
                        getline(myfile, line); // lines to skip
                }
```

```
                for (unsigned int i(0); i < section; i++)
                // only for the size of the section
                {
                        getline(myfile, line);
                        vector<string> parts;
                        // ... and process out the 4th element in the CSV.
                        split(line, ',', parts);
                        if (parts.size() != 3) {
                                cerr << "Unable to process line " << i
                                                << ", line is malformed. "
                                                << parts.size()
                                << " parts found."<< endl;
                                continue;
                        }

                        // Add this value to the account running total.
                        runningTotal += atof(parts[2].c_str());
                }
                myfile.close();
        }
        else {
                cerr << "Unable to open file";
                return NULL;
        }

        double* rT = (double*) malloc(sizeof(double));
        *rT = runningTotal;
        return rT; // return the total of this section
}

/*
 * Main entry point.
 *
 * This program expects one argument, the name of the file to process.
 *
 * argc - the number of command-line arguments.
 * argv - command line arguments array.
 */
int main(int argc, char* argv[]) {
        // Check the number of parameters
        if (argc < 2) {
                // Tell the user how to run the program
                cerr << "Usage: " << argv[0] << " filename" << endl;
                /* "Usage messages" are a conventional way of telling the user
                 * how to run a program if they enter the command incorrectly.
                 */
                return 1;
        }

        // Open the file again and process line by line.
        string line;
        ifstream myfile(argv[1]);
        filename = argv[1];
        double runningTotal = 0.0;

        // Count the number of lines in the file.
        // New line characters will be skipped unless we stop it from happening:
        myfile.unsetf(ios_base::skipws);
```

```
        line_count = std::count(std::istream_iterator<char>(myfile),
                        std::istream_iterator<char>(), '\n');

        // Handle files that do not have an ending newline character
        myfile.clear();
        myfile.seekg(-1, ios::end);
        char lastChar;
        myfile.get(lastChar);
        if (lastChar != '\r' && lastChar != '\n')
                line_count++;

        // Reset the position in the file and the stream attributes.
        myfile.setf(ios_base::skipws);
        myfile.clear();
        myfile.seekg(0, ios::beg);


        // Read in, line by line...

        pthread_t threads[4];
        int rc;
        int i;
        for(i=0; i < 4; i++)
        { // 4 threads are created
                rc = pthread_create(&threads[i], NULL,
                ReadLines, (void *)i);
        // passing to the ReadLines function and the thread id
                if (rc){
                        cout << "Error: unable to create thread," << rc << endl;
                        exit(-1);
                }
        }

        void* thread_result; // to store the result

        for(i=0; i < 4; i++)
        {
                pthread_join(threads[i],&thread_result);
                // wait for the thread to finish before getting result
                runningTotal = *(double*)thread_result + runningTotal;
                // add the result of that section to the running total
        }


        cout << setprecision(2) << fixed << runningTotal << endl;

        pthread_exit(NULL);

        return 0;
}
```