

1 Task 1

The shared memory approach bases around splitting the task using a single machine, such as in Figure 1. The MPI approach is still about splitting the data, but instead over one machine, it can be split across multiple physical machines, see Figure 2. So the program needs to change from current version where the lines are split between multiple threads. To where the lines are split between multiple nodes.

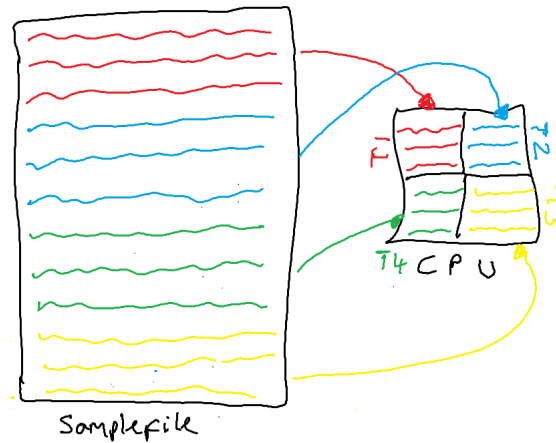


Figure 1: How the file is processed using the shared memory approach.

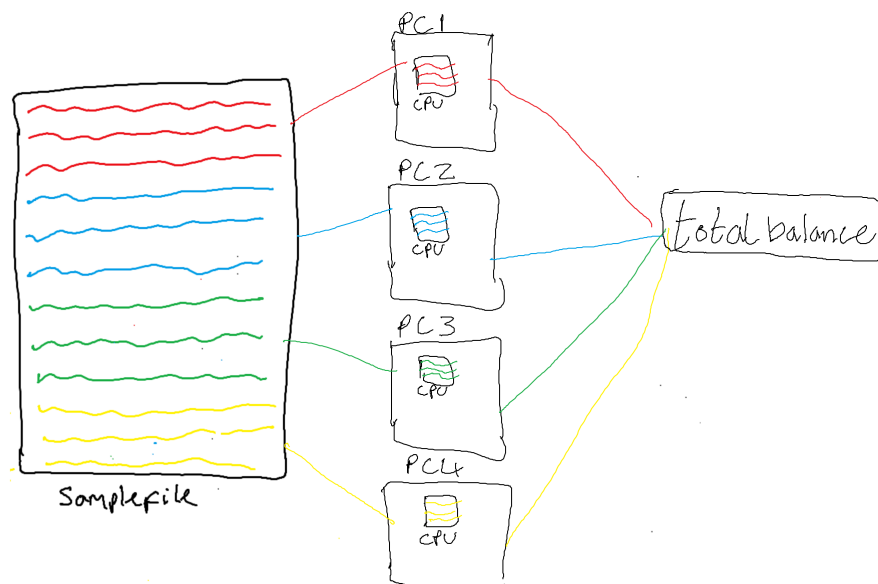
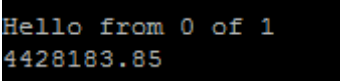


Figure 2: How the file is processed using the mpi approach.

2 Task 2

I firstly loaded the correct modules to be able to compile the program. To do this, in Putty I typed 'module purge' which resets the list of loaded modules. Followed by 'module load compiler/gnu/auto' to load for the gnu compiler. Then 'module load mpi/openmpi/1.8.5' to load the MPI libraries.

I then edited the MakeFile to set the new compiler, by adding 'CXX = mpicxx' at the top of the file. I then added the sample file code to the balance reader's main method to initialize MPI and test that it



```
Hello from 0 of 1
4428183.85
```

Figure 3: Output after running my sample file

works.

Upon running the program I obtain Figure 3.

2.1 MakeFile

```
CXX = mpicxx

CXXFLAGS =      -O2 -g -Wall -fmessage-length=0

OBJS =          balance_reader.o

LIBS =          -lpthread

TARGET =        balance_reader

$(TARGET):      $(OBJS)
                $(CXX) -o $(TARGET) $(OBJS) $(LIBS)

all:            $(TARGET)

clean:
                rm -f $(OBJS) $(TARGET)
```

2.2 Balance Reader

```
//=====
// Name       : balance_reader-exemplar.cpp
// Author      : Gray, Cameron C.
// Version     : 1.0
// Copyright   : ICP-3029, School of Computer Science, Bangor University
// Description  : PThread Reader - 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 <pthread.h>
#include <math.h>
using namespace std;

vector<int> positions;

struct process_data {
    int start;
    int offset;
    char* filename;
    pthread_t thread_h;
```

```

};

/*
 * Split a std::string based on a supplied delimiter.
 * s - the string to split.
 * c - the delimiter 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()));
    }
}

/*
 * The function to be used for each thread.
 * data - the void* pointer to the supplied arguments.
 *
 * Returns a pointer to this thread's answer.
 */
void* process(void* data) {
    process_data* p = (process_data*) data;

    long* runningTotal = (long*) malloc(sizeof(long));
    *runningTotal = 0;
    ifstream myfile(p->filename);
    if (p->start > 0) {
        myfile.unsetf(ios_base::skipws);
        myfile.seekg(p->start + 1, ios::beg);
        myfile.setf(ios_base::skipws);
    }

    string line;
    // Read in, line by line...
    if (myfile.is_open()) {
        for (int current_line = 0; current_line < p->offset; current_line++) {
            vector<string> parts;
            getline(myfile, line);

            // ... and process out the 3rd element in the CSV.
            split(line, ',', parts);
            if (parts.size() != 3) {
                cerr << "Unable to process line" << current_line
                    << ", line is malformed." << parts.size()
                    << " parts found." << endl;
                continue;
            }
            // Add this value to the account running total.
            *runningTotal += (long) roundf(atof(parts[2].c_str()) * 100);
        }
    }
}

```

```

    }
    myfile.close();
} else {
    cerr << "Unable to open file";
    return NULL;
}
pthread_exit(runningTotal);
}

/*
 * 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[]) {

    MPI::Init(argc, argv);
    int procs = MPI::COMM_WORLD.Get_size();
    int rank = MPI::COMM_WORLD.Get_rank();
    cout << "Hello from " << rank << " of " << procs << endl;
    MPI::Finalize();

    // Check the number of parameters
    if (argc < 3) {
        // Tell the user how to run the program
        cerr << "Usage: " << argv[0] << " _filename _number-of-threads" << endl;
        /* "Usage messages" are a conventional way of telling the user
         * how to run a program if they enter the command incorrectly.
         */
        return 1;
    }

    int number_threads = atoi(argv[2]);
    if (number_threads < 0 || number_threads > 100) {
        cerr << "Usage: " << argv[0] << " _filename _number-of-threads" << endl;
        cerr << "The number of threads must be between 1 and 100." << endl;
        return 1;
    }

    // Open the file again and process line by line.
    ifstream myfile(argv[1]);

    // Count the number of lines in the file and store the positions to allow
    // easy seeking.
    // New line characters will be skipped unless we stop it from happening:
    myfile.unsetf(ios_base::skipws);

    unsigned 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++;

    // Handle threads
    int n = line_count / number_threads;
    if (n < 1) {
        n = 1;
        number_threads = line_count;
    }

    int m = line_count % number_threads;

    myfile.clear();
    myfile.seekg(0, ios::beg);
    istream_iterator<char> first = istream_iterator<char>(myfile);
    istream_iterator<char> last;

    // Store the positions where each thread needs to start.
    int j = 0, k = m;
    positions.push_back(-1);
    while (first != last) {
        if (*first == '\n') {
            j++;
            if ((j == (n+1) && k > 0) || (j == n && k <= 0)) {
                k--;
                positions.push_back(myfile.tellg());
                j = 0;
            }
        }
        ++first;
    }

    // Close the file.
    myfile.close();

    // Handle distributing processing. The first m threads will
    // be asked to handle 1 extra line to best handle remaining
    // loads.
    int i;
    j = m;
    process_data proc[number_threads];
    for (i = 0; i < number_threads; i++) {
        proc[i].filename = argv[1];
        proc[i].start = positions[i];
        if (j > 0) {
            proc[i].offset = n + 1;
            j--;
        } else {
            proc[i].offset = n;
        }

        pthread_create(&proc[i].thread_h, NULL, &process, &proc[i]);
    }

    // Collect the subtotals from each thread, reduce it to a single
    // variable.
    long* returned;
    long runningTotal = 0;

```

```
    for (i = 0; i < number_threads; i++) {  
        pthread_join(proc[i].thread_h, (void**) &returned);  
  
        runningTotal += (long) *returned;  
        free(returned);  
    }  
  
    // Print out the total balance.  
    cout << setprecision(2) << fixed <<  
        (double)((double)runningTotal/(double)100) << endl;  
  
    return 0;  
}
```