School of Computer Science: assessment brief

Module title	Parallel Computation
Module code	COMP3221
Assignment title	Coursework 2
Assignment type and description	MPI Programming Assignment
Rationale	To implement a calculation on a distributed memory system using MPI, including a binary tree using point-to-point communication. Also, to perform timing runs for parallel scaling and to interpret your results.
Guidance	See overpage
Weighting	20%
Submission deadline	2pm, Thursday 20th March
Submission method	Gradescope
Feedback provision	Marks and comments returned *via* Gradescope
Learning outcomes assessed	Apply parallel design paradigms to serial algorithms. Evaluate and select appropriate parallel solutions for real world problems.
Module lead	David Head

1. **Assignment guidance**

   This coursework specification is for school Unix machines only, including the remote access `feng-linux.leeds.ac.uk`. We cannot guarantee it will work on any other environment.

   Someone is writing some statistics software that reads in a list of floating point numbers from a file, and calculates the mean and variance of this data set. Given that the data set is large, they want to improve performance by using MPI to perform calculations in parallel. You are tasked with implementing these calculations in parallel using MPI.

   The mean and variance are defined as follows. Suppose you have a data set with N floating point numbers x_i. Then the mean μ and variance σ^2 are defined as

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \quad ; \quad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N} (x_i - \mu)^2 \quad .$$

   Equivalently, in serial code, with indexing starting from 0, these calculations can be performed as

```
float sum = 0.0f;
for( int i=0; i<N; i++ ) sum += x[i];
float mean = sum / N;

float sumSqrd = 0.0f;
for( int i=0; i<N; i++ ) sumSqrd += (x[i]-mean)*(x[i]-mean);
float variance = sumSqrd/N;
```

   Note that you must first calculate the mean before calculating the variance.

   The following files are available from Minerva to get you started with this assignment:

`cwk2.c`	:	Starting point for your solution. Reads in the data file onto rank 0 as an array of `float`s. Also includes a serial check, which shows how the mean and variance are calculated.
`dataSet.txt`	:	Data to test your code and perform the timing runs.
`cwk2_extra.h`	:	Routines to load the data file and output the result. **Do not modify this file**; it will be replaced with a different version for assessment.
`readme.txt`	:	Use to provide results of your timing runs, and your interpretation (see below).
`makefile`	:	A simple makefile (usage optional).

   Note that, when you first compile the provided code, you will most likely get a compiler warning about the unused variable `localSize`. You can ignore this – the warning will go away once you start to use `localSize` in your solution.

For this assessment, the executable should always be launched with number of processes[1] that is a power of 2, *i.e.* 1, 2, 4, 8, ..., but no greater than 256. The starting code provided (see below) already checks for this.

2. **Assessment tasks**

   You will need the material up to and including Lecture 11 for this assignment.

   Currently the provided code outputs 0 for both the mean and variance, but some non-zero values for the serial check. Your goal is to implement MPI calculations that reproduce the serial result. Note that there may be small differences in the results of the parallel and serial calculations, even if your implementation is correct; this is explained in Lecture 11. The autograder takes this into account.

   More specifically, there are 4 tasks for this assignment:

   1. Implement MPI code that calculates the mean using as many processes as are made available to you. Your final answer should be left in the variable `mean` which is declared in `cwk2.c`.
   2. Communicate `mean` to all processes, so they can use it as part of their variance calculations. Although this is easily achieved using collective communication routine(s), you should instead implement a binary tree using point-to-point communication. You **must** use either of the two binary trees given in Lecture 11 (but 'upside down')[2]; you may find the second easier as the indexing is more straightforward. You may assume that the number of processes is a power of 2 – the provided `cwk2.c` already checks for this.
   3. Now that every process has the mean, you should implement a parallel calculation of the variance, leaving the answer in the variable `variance` that is declared in `cwk2.c`. **For the purposes of this coursework, your solution should not calculate the mean and variance at the same time**; instead, you should calculate the mean in parallel, distribute the mean to all process, and then calculate the variance in parallel using this mean.
   4. Once you are happy your solution is working, you should perform some timing runs on the provided `dataSet.txt`, and insert the results into the table in the given `readme.txt` file. You should also fill in the two sections at the end of this file, which ask you to explain your results. Please read the instructions in `readme.txt` for details; also see below. The provided `cwk2.c` file already includes the relevant MPI calls and outputs the timing of the parallel calculation.

   For task 2, you may like to first distribute the mean using collective communication routine(s), then revisit this when the rest of your solution is working to replace it with a binary tree. You may find the following useful:

[1] With OpenMPI you can force more processes than you have cores by using the `--oversubscribe` option when launching your executable with `mpiexec` or `mpirun`.

[2] These two forms of binary tree utilise processes multiple times during reduction and therefore more uniformly distribute the load, compared to other binary trees which only utilise each process once, and therefore have more idle time and are less efficient. Therefore, for full marks, your binary tree must be one of the ones covered in Lecture 11.

`1<<n`	:	Evaluates as 2^n.
`int lev=1; while(1<<lev<=p)lev++;`	:	Finds the number of levels `lev` in a binary tree with `p` leaf nodes.

If you do not implement the binary tree, you will lose marks but can still achieve a first-class grade.

For task 4, you need to fill in the table in `readme.txt`, launching your executable on the number of machines, and the total number of processes `<p>`, specified in the first two columns of the table. For one machine you should launch as normal,

```
> mpirun --oversubscribe -n <p> ./cwk2
```

with `<p>` the total number of processes specified in the table. For two machines, you will first need to create a file called *e.g.* `hosts` with the hostnames of two machines in Bragg 2.05 or similar; for example,

```
> cat hosts
uol-pc-055409
uol-pc-055410
```

then launch as

```
> $(which mpirun) -machinefile hosts --oversubscribe -n <p> ./cwk2
```

See Lecture 8 for more details. Note you can launch your code on two machines in Bragg remotely by first logging into `feng-linux.leeds.ac.uk` .

The `readme.txt` file also asks for the number of cores on each machine. One way to find this is to call the `helloWorld` example from Lecture 8 (or, indeed, Lecture 2).

3. **General guidance and study support**

   If you have any queries about this coursework, visit the Teams page for this module. If your query is not resolved by previous answers, post a new message. Support will also be available during the timetabled lab sessions.

4. **Assessment criteria and marking process**

   Your code will be checked using an autograder on Gradescope to test for functionality. Staff will then inspect your code then allocate the marks as per the mark scheme (see below).

5. **Submission requirements**

   You should upload the files `cwk2.c` and `readme.txt` to Gradescope using the portal for this coursework. This will test your submission for correct functionality, and return immediately with the results. If any of the checks were failed, modify your code and re-submit until all checks are passed. Late penalties will be calculated from the date and time of your final submission.

   The autograder will also scan your `readme.txt` file to see if your speed-up calculations are correct. This expects your file to contain one table with exactly 4 columns of numbers. If it fails, then (a) check that your table has exactly 4 columns, all filled in with values; and (b) there are no non-numeric characters in the table (such as `s` or `secs` after the times), which causes the value to be read as zero.

Although most of you will only need to upload `cwk2.c` and `readme.txt`, if you have added new files, you can also upload them and your modified `makefile`. Ensure all submitted files are in one, flat directory. If doing this it is important to make sure your modified makefile still works on the autograder.

Do not modify `cwk2_extra.h`, **or copy any of the content to another file and then modify**, as this file will be overwritten with an alternative version for assessment.

6. **Academic misconduct and plagiarism**

   Academic integrity means engaging in good academic practice. This involves essential academic skills, such as keeping track of where you find ideas and information and referencing these accurately in your work.

   By submitting this assignment, you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.

   If you use material from outside the module material available on Minerva, include reference(s) in your comments.

   There is a three-tier traffic light categorisation for using Gen AI in assessments. This assessment is **amber** category: AI tools can be used in an assistive role. Use comments in your code to declare any use of generative AI, making clear what tool was used and to what extent.

   Code similarity tools will also be used to check for collusion.

7. **Assessment/marking criteria**

   There are 20 marks in total.

5 marks	:	Correct functionality of mean and variance calculation. Correct calculation of the parallel speed-ups.
6 marks	:	MPI implementation of mean and variance calculations.
6 marks	:	Binary tree distribution of the mean to all processes.
3 marks	:	Interpretation of the results of the timing runs.