School of Computing, University of Leeds

COMP3221
Parallel computation
Coursework 2: Distributed memory parallelism with MPI

Deadline: Monday 21st March, 10am.

If you have any queries about this coursework, visit the Microsoft Teams page for this module. If your query is not resolved by previous answers, post your query as a new conversation.

This piece of work is worth **20%** of the final module grade.

Learning objectives

- Use collective communication to distribute the problem and accumulate the answer.
- Implement a binary tree using point-to-point communication.
- Perform timing runs for parallel scaling and interpret your findings.

Task

Your task is to implement an MPI-C program that performs matrix–vector multiplication in parallel. That is, given the $N \times N$ matrix A and the $N-$vector $\mathbf{x}$ defined on rank 0, you need to calculate $\mathbf{b} = A\mathbf{x}$, *i.e.*

$$b_j = \sum_{i=1}^{N} A_{ij} x_j \quad ,$$

in parallel, and ensure the full $N-$vector $\mathbf{b}$ is on rank 0 at the end of the calculation.

For convenience, rather than store the matrix A as a two–dimensional array, it is instead stored as a one–dimensional array of size N^2. This ensures the rows of the matrix are stored adjacent in memory, which makes the use of collective communication easier. For this flattened array, the matrix element at row `row` and column `col` is indexed as `A[row*N+col]`, and the serial code that performs the multiplication is[1]

```
int row, col;
for( row=0; row<N; row++ ) {
        b[row] = 0.0f;
        for( col=0; col<N; col++ )
                b[row] += A[row*N+col] * x[col];
}
```

[1] As for vector addition in the lectures, indexing starts from 1 in mathematical notation but 0 in C-code.

You are provided with code that initialises the matrix `A` and solution vector `b` on rank 0, with the problem size N specified as a command line argument. It also allocates memory for the vector `x` on all processes, and the per–process matrix `A_perProc` and per–process `b_perProc`. You will need to make sure each process receives a copy of the full vector `x`, and a partition of `A` by rows. Each process will then calculate the multiplication for their portion of the matrix, leaving the answer in `b_perProc`. The `b_perProc` for each process will then need to be combined into the final solution `b` on rank 0.

Once you have this working using collective communication, you should then implement an alternative distribution of `x` to all processes that uses point-to-point communication in a binary tree pattern. This method should be automatically called if the number of processes `numProcs` is a power of 2; if it is not a power of 2, your code should continue to use your original version. Your submitted code should therefore include two versions of the distribution of `x`, each being called depending on the number of processes specified when launching with `mpiexec`.

One you have the final version of your code, perform timing runs to determine the parallel execution time as the number of processes is varied (the provided code already outputs this time). The combination of numbers of processes and nodes is given in the file `readme.txt`, and you should insert your results and discussion into this file, and include it when submitting. You should also interpret your results in terms of your understanding of MPI and the architecture you used for these timing runs.

Guidance

The provided files are:

`cwk2.c`	:	Starting point for your solution.
`cwk2_extras.h`	:	Routines to allocate, deallocate, and initialise the problem. **Do not modify this file or avoid calling its functions**.
`readme.txt`	:	Use to provide results of your timing runs, and interpretation.
`makefile`	:	A simple makefile (usage optional).

It is recommended that you proceed incrementally, testing your code after each stage of your calculations. For the binary tree, it is useful to insert print statements showing which rank each process is sending to or receiving from, to ensure that all sends have corresponding receives.

Point–to–point communication was covered in Lecture 9 and collective communication in Lecture 10. For the binary tree, you may like to consider an upside–down version of second variant considered in Lecture 11, as it is easier to code than the first example.

You may find the following useful.

`1<<n`	:	Evaluates as 2^n.
`if( n && ((n&(n-1))==0) )`	:	Evaluates as `true` if `n` is a power of 2.
`int lev=1; while(1<<lev<=p)lev++;`	:	Finds the number of levels `lev` in a binary tree with `p` nodes in the first layer.

You are required to use `cwk1_extras.h` unaltered, but are free to add new routines to `cwk1.c`. It is envisaged that you will only edit this one file, but if you do add more files, ensure you amend the makefile following the instructions it contains.

Marks

There are 20 marks in total:

4 marks	:	Distribution of matrix `A` and solution vector `b` between processes.
8 marks	:	Distribution of `x` to all processes, using a binary tree when the number of processes is a power of 2. Max. 2 marks if there is no binary tree version.
4 marks	:	Correct parallel implementation of matrix–vector multiplication.
4 marks	:	Results from timing runs and interpretation.

Note that you can achieve a first class mark without implementing the binary tree, in which case your code should use the same method of distribution of `x` for *all* values of `numProcs`.

All submissions will be compiled and executed on a School Linux machine similar to those in labs. If you alter the makefile for any reason, ensure it still works on a School machine before submitting.

Submission

Your are required to submit `cwk1.c` and `readme.txt`. If you have added extra files, ensure they are all in a flat directory (*i.e.* do not use subdirectories) and that the makefile has been suitably amended. Do not change the executable name. Submit all files including the new makefile as a single archive (`.tar.gz` or `.zip` formats **only**).

Do not modify `cwk2_extras.h`, **or copy any of the content to another file and then modify**. This file will be replaced with a different version as part of the assessment, so it is imperative that your code still calls these routines.

Disclaimer

This is intended as an individual piece of work and, while discussion of the work is encouraged, what you submit should be entirely your own work. Code similarity tools will be used to check for collusion, and online source code sites will be checked.

Ensure you retain the receipt for your submission as you may be asked to produce it at a later date in the event of a disputed submission time.

The standard late penalty of 5% per day applies for work submitted after the deadline.