**COMP4300/8300:   
Parallel Systems**

**Semester 1 2015**

**Assignment 1**

**Distributed Matrix Multiplication Using MPI**

**Deadline: 09:00 on Tuesday 7 April**

**REVISED (due to Raijin problems) Deadline: 09:00(am) on Thursday 9 April**

The late penalty is 5% per working day to a maximum of 10 days or until when the assignment is returned, whichever is the least)   
(Please report any errors, omissions and ambiguities to the course lecturer)  
(Amendments to this document from now on will be marked in red)

This assignment is worth 20% of your total course mark.

**Submission**

This assignment must be submitted electronically. You will submit a tar file containing a number of other files. This is done using the following command (according to your enrollment):

* submit comp4300 ass1 myass1.tar
* submit comp8300 ass1 myass1.tar

(You run this command on partch.anu.edu.au or any of the other Linux machines that are in the CSIT labs)

**Objectives**

* Implement a distributed algorithm using message passing.
* Model the computation and communication costs of the algorithm.
* Measure the performance of the implemented code, and compare against the model.

**Description**

**Setup**

The file [ass1.tar](http://cs.anu.edu.au/courses/COMP4300/Ass1/ass1.tar) contains template files for the files that you will submit. It also contains an example test harness harness.c, and a Makefile which should be used to compile the programs. Copy these files into a local directory before you begin work on your programs. You can compile the system with the simple command make. (The code uses the Intel MKL library, so you may need to do "module load intel-mkl" for the code to link.)

The test harness calls a number of functions in turn to perform distributed matrix multiplication. After each call, it compares the resulting matrix C with the result of a local DGEMM. The functions are:

* mult\_replicated
* mult\_summa

Your task is to implement these functions.

You can compile the harness code using the Makefile provided. On the NCI system, we will use the [Intel Math Kernel Library](http://software.intel.com/en-us/intel-mkl) implementation of the [BLAS](http://www.netlib.org/blas/) and [LAPACK](http://www.netlib.org/lapack/) operations. On Raijin, load MKL as follows:

module load intel-mkl

The provided ass1.c contains a naïve implementation of matrix multiplication as follows:

for (i=0; i<m; i++) { for(j=0;j<n;j++) { C(i,j) = 0.0; for(l=0;l<k;l++) { C(i,j) += A(i,l) \* B(l,j); } } }

**Part 1: Replicated Data**

1. Start by measuring the performance of the C code given above and comparing it to the performance of BLAS DGEMM for a range of different matrix dimensions.
2. Implement a distributed dense matrix multiply C = A x B using replicated data. Options include: broadcasting an entire matrix, sending entire rows or columns of a matrix to each process, inner or outer product formulation, etc.

You must implement the following C functions:

int mult\_replicated(int m, int n, int k, double\* a, double\* b, double\* c);

The test harness program calls these functions. The inputs are:

m, n, k

the matrix dimensions

a

the matrix A at process 0, dimensions m × k, i.e. m rows by k columns in column-major format

b

the matrix B at process 0, dimensions k × n

c

the matrix C at process 0, dimensions m × n (uninitialised)

Within each of this routine you should distribute the data, evaluate part of the matrix multiplication at each process, and combine the results so that process 0 has a complete copy of the final matrix.

Multiplication of subblocks of the matrix may be performed by a call to the BLAS subroutine dgemm. (The BLAS library call includes support for alpha or beta scaling factors that are non-unit value and transposed matrices - your code is not required to support this.)

1. Model the computation and communication time of your algorithm.

Show the costs separately for each phase of the computation. Is the algorithm communication or computation limited, or is there a transition point?

Use your model to predict the total execution time taken by your routine mult\_replicated for the following square matrix multiplications and process counts run on Raijin. Justify the choice of any parameters you might use as part of your model (eg bandwidths or flop rates).

|  |  |  |  |
| --- | --- | --- | --- |
| Predicted execution time and GFLOP/s for square matrix multiplication with replicated data | | | |
| **No. Processes** | **N** | **Time** | **GFLOP/s** |
| 1 | 1000 |  |  |
| 2 | 1000 |  |  |
| 4 | 1000 |  |  |
| 8 | 4000 |  |  |
| 16 | 4000 |  |  |
| 32 | 4000 |  |  |

1. Enhance the harness program to time a number of calls to mult\_replicated and report the minimum measured time. Now gather actual performance data for the same matrix dimensions and processor counts as above.

|  |  |  |  |
| --- | --- | --- | --- |
| Measured execution times on Raijin and GFLOP/s for square matrix multiplication with replicated data | | | |
| **No. Processes** | **N** | **Min Time** | **Max GFLOP/s** |
| 1 | 1000 |  |  |
| 2 | 1000 |  |  |
| 4 | 1000 |  |  |
| 8 | 4000 |  |  |
| 16 | 4000 |  |  |
| 32 | 4000 |  |  |

1. Compare the measured values with the predicted values. Explain any difference. (Provide additional data points if required.)
2. Explore the performance of your code for a range of different non-square matrix sizes. (Only record data for up to 32 cores. Limit total memory usage per core at around 1GByte max.)

**Part 2: SUMMA**

1. Use the provided code for the [SUMMA algorithm](http://cs.anu.edu.au/courses/COMP4300/lectures/SUMMA-preprint.pdf) to perform distributed matrix multiplication.  
   The code for the SUMMA (function pdgemm described in the paper) is provided in the file [summa.c](http://cs.anu.edu.au/courses/COMP4300/Ass1/summa.c). It assumes that each place holds a block of the A and B matrices, and computes a block of the C matrix. You must implement the following C function:

int mult\_summa(int m, int n, int k, double\* a, double\* b, double\* c);

This should distribute data to the appropriate processes and then call pdgemm with the correct parameters. This requires that you determine the dimensions of the local blocks of A, B and C for each process, and create row and column communicators required by SUMMA. You may choose whatever default value you think is sensible for the panel size nb.

1. Model the computation and communication time for the algorithm provided.
2. Measure performance on Raijin as described in part 1 above. How does this compare with the cost model?

**Requirements**

Your code must be written using C and MPI, with calls to BLAS and LAPACK subroutines and system libraries. Calls to other libraries are not permitted.

You are required to submit a tar file (myass1.tar) containing the following:

* a *plain text* file called README that details all files within the tar file and provides instructions on how to build and run your code on Raijin
* A makefile that when executated will build your code
* File ass1.c, modified to inlcude the function definitions completed by you.
* A PDF file called WRITEUP.pdf containing:
  + a disclaimer with your name and student number stating what (if any) contributions from others (not including course staff) are in your submission. Note that significant contributions may require a revision of your final mark, as this is intended to be an individual assignment;
  + Your answers to all the questions given above.
  + Details of any notable deficiencies, bugs, or issues with your work.
  + Any feedback on what you found helpful in doing this work, or what from the course could be improved in helping you to carry out this work.

Your code should be written with good programming style (and so should not produce any warnings when compiled with the -Wall option!). It should be well commented, so as to enable the reader (this includes the tutor/marker!) to understand how it works. Identifiers should be meaningful and (unless their role is trivial) commented where declared, any constants should be named rather than hard-coded, and defensive programming should be used. The latter includes checks being made for function calls that can return an error status; once detected, a message should be generated with a call to the system error function perror() and the process should exit with a non-zero status. It also includes using assert() to perform other kinds of checks, and the freeing of any malloced data before (normal) exit (this permits the mcheck library to detect any buffer overwrites).

When running jobs on the Raijin system, please use the **normal** queue rather than the express queue, as the express queue is charged at 3x the standard usage units. Please be mindful that you share the allocation with all other students in this course!

**Marking Scheme**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Section** | **Implementation** | **Model** | **Measurement** | **Total** |
| Replicated data | 5 | 3 | 3 | 11 |
| SUMMA | 5 | 2 | 2 | 9 |
| **Total** | 10 | 5 | 5 | 20 |