CS160 – Winter 2018 – Programming Assignment #3

Due 1159pm, Feb 10, 2018 (Saturday)

The goal of this assignment is to parallelize is to start to get some understanding of performance (both computation and messaging). You will also be writing a master-worker program from matrix multiply.

You will be writing short "reports" for various aspects. All of your timings MUST RUN ON TSCC.

Any reference to floating point or double means to use `double`

PART 1

mm.c

You are being given a serial code, called mm.c, that creates two N x N matrices and multiplies them together a number of times and gives the average execution time for this. It is invoked as

```
./mm <N> <trials>
```

If you look at the code for mm, it is calculating C = A x B in standard matrix multiplication. All matrices are N x N. The entries of A and B are randomly-generated.

Task 1.1

Modify mm.c to support multiplying (N x M) x (M x P) matrices. It's new command line should be

```
./mm <N> <M> <P> <trials>
```

This modification should compute C = A x B, where A is NxM, B is MxP and the result C is NxP. The dimension requirements are needed so that the matrix multiplication is well-defined. You will need to validate that you are performing this new computation correctly.

Task 1.2

Create a graph with an appropriate title, labeling of axes, the plots the time it takes to multiply N x N matrices where N ranges from 16 to 512 in steps of 4 (e.g. 16, 20, 24, etc). Use 100 trials for each problem size. On the graph, label the time and approximate Floating point rate (Flops). Use $2*N^3$ to estimate the number of floating point operations for a given matrix multiply.

When asking for a node allocation on TSCC use "`-l nodes=1:ppn=1`" to be allocated a single core on a node of TSCC.

Task 1.3

Write a short (single page including graphic) that shows the performance of Task 1.2. In English, describe when you ran the job, what node of TSCC was used and if there was any significant difference in estimated floating point rate at different matrix sizes. If you observe a difference, you to give a reasonable explanation for the changes. Call this one page mini-report **task1.pdf**

Requirements/Hints/FAQ on Task 1

1. You do not have to check valid type of command-line arguments. We will test only with valid inputs. You do need to check for proper number of arguments.
2. Compile your code explicitly with gcc and use optimization option `-O3`. Do not use any other optimization.
3. You will be turning in one Makefile for the entire project for this part you must have a target called "mm" that compiles the final (Task 1.2 version) of mm.
4. You may use any graphic software in which you are comfortable. Excel, OpenOffice, GoogleSheets, Word, etc. to create your graph and mini-report

What you will turn in Task 1

1. Modified mm.c
2. Makefile (targets "mm" and "clean" are required (no quotes))
3. task1.pdf

Task 2

The goal of this task to "benchmark" message passing on TSCC. You are not given any starter code for this portion, but instead you are given a specification of the command-line arguments

Task 2.1

Create a code called msgbench, written in C, using MPI. It tests the speed in which two MPI processes can communicate with each other. There are numerous existing codes that do this (don't copy them), but it is useful to write a primitive one for yourself. It is called as follows

```
msgbench <msgsize> <trials> [bidirectional]
```

- msgsize is the size in bytes of a message to transfer
- trials is the number of roundtrips to make to determine how fast data can be moved
- [bidirectional] is an optional argument. If given, both processes should simultaneously send message of size <msgsize> to each other.

Its output should look similar to the following (on a single line):

```
4063232 bytes (500 trials) 0.033780 time 240570278.271166 Bytes/s
```

And

```
4063232 bytes (500 trials) 0.033780 time  23770278.382432 Bytes/s
(bidirectional)
```

We are NOT grading your output format. It should be informative, but the specific format is unimportant. You might want a different format (e.g. CSV for importing into excel, etc.). It's not difficult to take the above output and turn it into CSV.

In the unidirectional case:

1. Rank 0 creates a send buffer of at least msgsize bytes.
2. Rank 0 initializes this buffer with random data (not 0!)
3. Rank 0 starts a timer

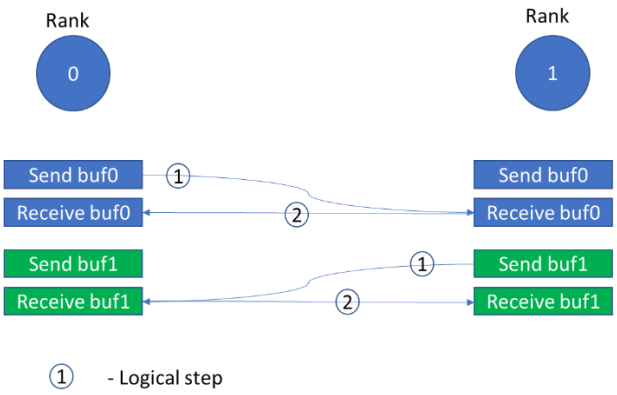
4. It sends the buffer to rank 1
5. Rank 1 sends back all the data that it just received
6. Rank 0 stops the timer when it has received the data it sent.
7. For better timing, repeat the loop indicated by 4 and 5 for n trials. Make sure that Rank0 doe does not send data until it has received the data from the previous trial. Compute the average time over n trials

Further discussion of Task 2.1

Benchmarking (well) is actually quite an art form and this isn't an exercise in getting the best benchmark. It's really about finding out what performance you can get writing your own programs. We're going to use this in the Task 3.

Some things to think about when designing your code

- The cpu timer we're using isn't highly accurate. You will need many trials at the smaller sized messages to even register a time change
- This is a case where asynchronous sends are really much easier. You want both sides to be able to send a message at the same time they are receiving without deadlock for any message size.
    - You need to be careful here that you have fully received what you expect
    - You further need to think about the bidirectional test. For example, rank 1 must receive all of the message 0 is sending it, BEFORE it can send it back. The same is true for rank 0.
- It's good idea to have a version with a "validate" capability so that you can verify that the data you sent is the data you received.
- It's simpler of both ranks to create four buffers of equal size.

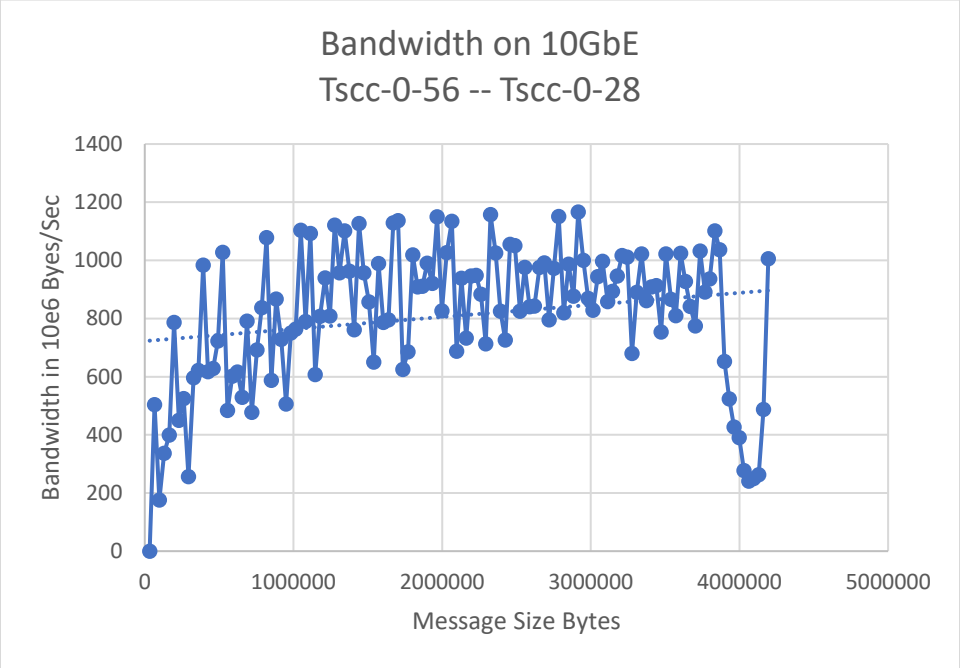


The picture above illustrates the idea of multiple buffers being allocated, but not all are utilized in all test cases. It simplifies the code to have this extra dummy space (and you won't lose points for taking this approach). Using asynchronous messaging for the forward (blue) case, Rank 0 can post a receive (receive buf0) and send the data in send buf0 to rank 1. If doing a bidirectional test, it can also post the receive for receive buf1. Once all of that is completed, it could (if a bidirectional test) send the receive buf1 back to rank 1. There are various waiting approaches (MPI_Waitany vs. MPI_Waitall) that are plausible. Pick something that makes sense to you.

This setup is often called “ping-pong” measurement. You measure the full roundtrip of data.

Task 2.2

Measuring the bandwidth achievable on TSCC. When you believe your code is working well, you should benchmark the network on TSCC to produce graphs that look similar (not identical) to the following



There are many things to observe about this graph. First it is noisy – tests were taken on a live system with many other users on it. Second, it shows the general trend of a fast rise in performance for smaller messages and a general leveling out at large messages. The particular network has a maximum performance of 1200x1e6 (1.2 Billion bytes/sec) and every now and then, the code used to generate the graph comes close. It also shows a large (unexpected dip) at around 4MB – this is probably transient congestion on the network.

Comments/Requirements about Task 2.2

- You are to produce graphs for the 10GbE network on TSCC and IB network
- TSCC has multiple networks, To test 10GbE on TSCC you must add the following flags to mpirun when running across two physical nodes

```
1. -mca btl self, tcp
2. -mca btl_tcp_if_include 132.249.107.0/24
3. -map-by node
```

- To test on Infinband, you do not use the first two parameters in your MPI run (include the last one), in your job submission script, you need the following directive
    a. #PBS -W NODESET:ONEOF:ibgroup0:ibgroup1:ibgroup3
- On the two physical nodes case, you should test the following

    a. You should measure bandwidth starting at 32KB (32768) and incrementing 32KB until you hit a maximum of 4MB (41904304) bytes (you may want slightly different ranges when testing infiniband vs. Ethernet)
- You are to run three cases, and create three different graphs: two of the graphs have MPI processes on two different physical nodes using two different networks. The third in which the MPI processes are on the same physical node The following are hints as to some "magic" that you need to give to something on TSCC to allocate nodes for your test
    a. `#PBS -l nodes=2:ppn=1`
    b. `#PBS -l nodes=1:ppn=2`
- On the single node case, you should measure bandwidth with the following mpirun parameter
    a. `-mca btl self,sm,tcp`
    b. You should measure bandwidth starting at 32 bytes with a maximum of 32768 in increments of 32 bytes
- Create a file called **task2.pdf** that has all of your graphs. Write a short paragraph or two that compares and contrasts the performance you observe among the cases. Also answer the question, Why are we measuring different message lengths in the various cases?
- We will not test your msgbench program with erroneous bad inputs in terms of format (we'll give you strings that convert to positive integers), but you must check for the correct number of arguments. Print out a helpful usage message when the wrong number of arguments are given.

What you will turn in Task 2

- msgbench.c (and any support files you wrote)
- Makefile (from above) (Targets "msgbench" and "clean" are required (no quotes))
- Task2.pdf

Task 3 - Bag of tasks (Master-Worker) version of Matrix Multiply

The final task of this project is to create a master-worker version of Matrix Multiply.

It will solve the equation C = A x B where the elements of A, B, C are <blocksize> x <blocksize> submatrices.

Your code must be invoked as follows:

```
mmmw <blocksize> <N> <M> <P> <output file>
```

- `<blocksize>` indicates the size of a submatrix
- `N` number of "rows" in A
- `M` number of "columns" in A and rows in B
- `P` number of "columns" in B

Block-based matrix multiply works just like regular matrix multiply, except when you multiply elements of the matrix, you must use matrix multiply.

Suppose that A is N x M, B = M x P, blocks where each block is k x k.

C(i,j) is also a k x k block. And it is:

Sum (A(i,q) x B(q,j)) q = 0,1,2,..M.

Task 3.1 Master-Worker Approach

Matrix multiply is well understood and highly optimized in a variety of settings. One of the problems with statically partitioning a large matrix across cluster nodes is that all nodes my not be of the same performance (e.g. differences in load, capability, network on various machines.). Noisy measurements from Tasks 1 and Tasks 2 give you an idea of what the real world looks like. You also have a feeling for how much time it takes to send a block of bytes using MPI to a different machine and on the same machine. You have an idea as to how long a matrix multiply of certain size takes on a single node. So now there is the balance – how much time is spent on communication vs. how much time is spent on computation.

The idea of master-worker is that a master has a job to perform ("matrix multiply") and the master can parcel out work of certain sizes to "workers" who perform a computation and then return the results. If a worker isn't given a big enough chunk of work to do, most of the time is spent sending data. If the chunk of work is too big, then a slow process can really make the entire process go slowly.

To make your task easier, the block size is uniform (and square). And the full A matrix is (N*k) X (M * k). elements. If you are given `mmmw 128 4 4 4 myoutput.txt`, then your multiplying 512x512 matrices using blocks of 128x128. In this case there are 4^3 matrix multiplies of 128 x 128 each. In other words, the master has 64 matrix multiplies that need to be carried out. The results of those matrix multiples need to be added properly to get the result. However, all the multiplies are independent of each other and can therefore occur in parallel.

Your job is design and implement your own master-work matrix multiply code. The master should be process 0 (and it does no multiply work, but it can be used to accumulate sums into the final result matrix). There must be at least 1 worker processes (hence mpirun -np 2 is a minimum).

How to get started on this task

The master can define a queue of work to do. Workers start up waiting for work to be given to them or told that there is no more work to be completed and they can exit. Note that workers can finish in any order and they do not need to know about each other. What's in the work queue? <blocksize> matrix multiplies. So, the first task, is to define how you want to label work into a subtask, how you want to communicate to between master and worker and how many subtasks at a time you will hand to workers. To make this concrete, in our 128,4,4,4 example above, the master creates the matrices A, B and initializes C to 0.00. There are 64 multiplies that need performing. Suppose you have 3 workers. Hand out the first 3, one to each worker. Wait. The first worker will return a completed multiply, add it to the existing data in C and then give that worker another unit of work. Keep repeating until you run out of work to do and the multiply is completed.

Initialization

The master allocates memory for A, B and C according to the command-line parameters. You might want to think about how to allocate the memory for these. It's convenient to be able to send a k x k block as a single memory buffer. This program is more easily written using asynchronous messaging.

A – Initialized to Random doubles

B – Initialized to Random doubles

C – Initialized to 0.0 in all elements

Output format in file and standard output

Define n = N x <blocksize>, m = M x <blocksize> and p = P x blocksize

The output file should should have the following format (ASCII Text)

Line 1: n,m,p

Next N lines : The rows of A where each element is separated by commas One row per line (m elements/line)

Next M lines: the rows of B, where each element is separated by commas. One row per line (p elements/line)

Next N lines: the rows of C, where each element is separated by commas. One row per line (p elements/line)

Your output file should have (2*N + M + 1) lines.

Please format your doubles with the %15.8f

Output on stdout for your program:

Your program can have any diagnostic output you desire. However within that standard output, it is required to have the following printed out only by rank 0 with keywords followed by followed by ":".

```
Parameters: blocksize, N, M, P, <# of workers>
Worker 0:  <# of block matrix multiplies performed>, <total time working in seconds>
Worker 1: <# of block matrix  multiplies performed>, <total time working in seconds>
…
Worker <m-1>: <# of block matrix multiplies performed>, <total time working in
seconds>
Total Time: time in seconds
```

Task 3.2 - Performance measurement of your code.

You are given some latitude on this measurement. Based on your measurements from Tasks 1 and 2. You should select problem sizes (block sizes, N, M, P) that show some of the characteristics that you discovered. Use 4, 8, and 15 workers on physically separate machines on TSCC AND on a single machine of TSCC. (Most TSCC nodes have at least 16 cores, hence 15 workers). When working in physically separate machines, request one core per node. Hence you will need 5, 9 and 16 cores to test in the distributed and same-node cases. This is a total of 6 tests. How you know what to use? The choice is up to you, but you might want to illustrate where mmmw does a poor job (perhaps slower than a single node), where it does a good job in performance (very good speed up). Where you can show off its ability to balance load across nodes of varying performance.

You need to write up your results into **task3.pdf**. This can be no longer than 3 pages (and it can be shorter – if you can say what you have to say in few pages, please do.) For each of the six cases we want information that includes:

1. The standard output from the test above (just the required parts)
2. The mpirun command used to run the test.
3. Which network was utilized on TSCC was utilized
4. A short (one or two sentences) description of what you expected to see when you chose this particular configuration to test
5. The speedup (or slowdown) that you actually measured (use the number of workers for the number of processors)
6. The calculated efficiency of the test
7. A short (one or two sentences) conclusion of what you learned about this particular testing configuration.

Please note, the writeups are not supposed to be long and involved. They need to be readable, but you won't be graded on grammar.

What you will turn in Task 3

- mmmw.c (and any support files you wrote)
- Makefile (from above) (Targets "mmmw" and "clean" are required (no quotes))
- Task3.pdf

Other Requirements

- One Makefile for all three tasks
- No large files (e.g. no output files from mmmw in your repository)
- We're not checking for correct *type* of arguments, but *will check for correct number* of arguments for each programming
- Reasonable commenting, as usual.
- Text in your writeups should be at least 11 point in a readable font (Times Roman, Arial, Calibri) with line spacing no more dense than how this document (the assignment document) is written (in Word using 1.08 spacing). All Margins must be at least 1". Page size must be US standard 8.5" x 11". Readers will be instructed to give you 0 points on any writeup that has spacing/fonts that are similar to:

  > This is very tightly packed text with a too small font. It's considered unreadable. If a reader sees your writeups using this type of spacing and/or small font you will get 0 points on the writeup. NSF/NIH have similar rules for their grants

What to turn in

- See the sections above.

Turning your program

It will be similar to PR2.

What you will be graded on

- Correctness of mmmw output. We'll run it
- Modified mm runs and reports reasonable performance results
- Msgbench runs and reports reasonable performance results
- All codes handle # of arguments correctly with usage statements
- Indentation – Your code must be indented. Choose a style and stick with it.
- Comments – You must comment your code in a reasonable way. At a minimum, your name, ucsd email address, student ID must appear in comments. All your routines should be commented with a brief description of what the routine does, what it returns and a description of the parameters.
- Your task[123].pdf writeups. We'll be looking for reasonable explanations. Your total writeups should not exceed 5 pages. Don't take it as a challenge to put in every detail, but you need to be able to crisply explain what is happening. You won't lose points if a 3 page write-up is only 2 or 2.5 pages and "got the job done". Page counts INCLUDE figures.
- Your file output should be written by process 0.

Some questions that you do not need to answer formally but are worth thinking about

- You chose a particular way to hand out work to workers? Is there another approach that you might have taken to reduce the number of times you resent a block? Sketch it on a piece of paper.
- There are numerous networks on TSCC, what are they?
- The -mca parameters look a little cryptic, what in particular does -mca btl self,sm,tcp mean?
- What is the purpose of `-map-by node` parameter to mpirun?
- What happens if a node becomes really really slow while it has a work task .. what could you do to address this?
- What is superscalar speed up? Is there a configuration in which you might see this phenomenon?