ECE382N-10: Parallel Computer Architecture
Fall 2023
Derek Chiou

Lab 1: Matrix Multiply using MPI and Cilk

Due 9/25/2023 at 2:59PM (right before class)

Please submit the lab through Canvas as a tarball. The tarball should be named as follows.

Lab1_LastName1_LastName2.tgz

The tarball directory structure should be as follows:

Lab1Report_LastName1_LastName2.pdf

/LabReport all the source for your lab report

/Cilk your Cilk solution code and executable.

/MPI your MPI solution code and executable

Please submit lab through Canvas. Include a PDF of a brief lab writeup, stating your names, who did what in the lab, and a brief description of our solutions, along with the document source (plain text, Latex or Microsoft Word.) Please also send any source code as specified within problems that require coding. Be sure to indicate your names as well as who did what if there was any division of labor. Please work together as much as possible as you will all be responsible for all the material.

Part 1: MPI
Implement a message passing matrix multiplication using MPI on the Lonestar6 cluster at TACC. The intention of the lab is for you to parallelize using MPI. Thus, only use the standard EPYC cores and not the GPUs. You should also not use the SSE or AVX instructions. Use lab1.tar.gz as a starting point. Please read the Lonestar6 user guide before attempting to use Lonestar6. Please add your TACC username in course schedule "TACC" sheet if you have not done so that we could give you tacc access.
Lonestar6 - TACC User Portal (utexas.edu)

Remember, Lonestar6 is a shared supercomputer and we have a limited number of hours on it. Each team should not consume more than four SUs, and ideally fewer. Be sure to watch your job submissions and set a proper time limit in your sbatch file, as we've seen cases where a submission runs for longer than intended.

We will test your solution on Lonestar6, so we would strongly recommend you get it to work there. You are, of course, welcome to develop it wherever you like.

Testing will be done on no more than 8 nodes (1K cores!) but will also be done on fewer nodes and thus your code should be able to handle any number of nodes (including non-powers of two). Your code should also be able to handle any data size and will be tested with multiple data of different sizes.

Please be considerate about running. Test on small configurations (2 nodes) in the ***development*** queue until you are sure it works functionally before performance tuning on larger machines. Otherwise, we will run out of time. Remember, I can see how much time each of you has used.

Your code will be linked with a gen_matrix.c file that will define the matrix. Of course, I will test with a different gen_matrix.c than the one that is provided to you in the tarball. A sequential example of gen_matrix.c along with code that uses it can be found in lab1.tar.gz on Canvas. Gen_matrix.c contains a gen_sub_matrix function (documented in the code) that will generate a sub-matrix for each process. The number of matrices that you multiply together will be returned by a call to init_gen_sub_matrix with the test_num as its sole argument. ***For simplicity, the test matrices will be the same size and square. The number of elements in one side of the matrix will be a multiple of the number of processors.***

You are allowed to call gen_sub_matrix at most once per process per argument matrix. You are allowed a total of (1/N * matrix_dimension * matrix_dimension) * (num_matrices + 3) of heap storage per processor and nothing substantial (i.e., no arrays) can be stored on the stack. Note that this means that you must figure out a way to map the input matrices and output matrix across all the processors. You cannot generate the entire matrices on each processor (otherwise, the problem gets easier.)

To make it easier for us to track how much memory you use, all heap objects, including arrays, in your code must call my_malloc to be allocated (in other words, no arrays to be allocated on the stack and don't use malloc, calloc, etc. directly.)

The executable that you deliver will be called test_mm and should take exactly three integer arguments, debug_perf, test_num, and matrix_dimension_size that are described below.

debug_perf: 0 indicates debug mode and 1 indicates performance testing mode. In debug mode, all argument matrices and the final result matrix will be printed in rows of numbers, each number on a single row separated by a space. Before each argument matrix print "argument matrix %d", where %d is from 0 to the number of argument matrices - 1 and "result matrix" before the result matrix. There should be no space between the title and the matrix, but one empty line

after each matrix. You must adhere to the memory space constraints, even in debug mode. In performance mode, print a single floating-point number that is the sum of all the elements in the final result matrix.

test_num: Passed to gen_sub_matrix. This is the test number that I will be running to functional test and performance test your code.

matrix_dimension_size: the size of one side of the matrix.

The fastest code for my select set of matrices (in absolute wall clock time) will be acknowledged in class and will be awarded extra credit.

Thus, your job is to mimic the precise functionality of the code found in lab1.tar.gz but, of course, much faster due to MPI parallelization.

This code is non-trivial, but will give you a good idea of how to write pure message passing code. We recommend starting earlier as, depending on what jobs are in front of yours, your jobs do not execute immediately on Lonestar6.

Hint: Think about the differences between blocking and buffered message passing.

What you hand in: A tarball that contains your code and a makefile that incorporates a gen_matrix.c, gen_matrix.h, my_malloc.c and my_malloc.h as in the original lab1.tar.gz tarball. The tarball should be named Lab1_LastName1_LastName2.tar.gz. The directory where your code resides should be named Lab1_LastName1_lastName2. If you have three people in your team, you will obviously have three last names in your filenames. The last names should be in alphabetical order. Only ONE member of the group should submit.

The makefile should contain the same targets as found in the original makefile including test_mm, run_debug, and run_performance. To test the correctness and performance of your code, I will first "make test_mm" to build your application, run in debug mode once to test the correctness of your result and then run in performance mode several times to test the performance of your application. The speed of your solution will depend on the average of run_performance.

To run the code, I will use the sbatch command on TACC and specify the number of cores that should be allocated by the TACC system. It is your job to determine the number of cores and use all of them using MPI_Comm_size.

Evaluation Criteria: I expect all of you to at least be able to deliver correct solutions on all inputs. I also expect that you will achieve some speedup. Speedup will be measured from your -n 1 (single core) run which must be reasonable. Reasonable will be determined by your classmates -n 1, a standard single-processor triply nested loop implementation and my reference implementation. So, don't make your single processor run slow to get better speedup. Do the best you can with data layout and

scheduling communication. Elegance and parallel performance relative to the others will be the criteria used for above average evaluation.

Here is what running MPI on TACC looks like.

Modify the mpicc script to change from the Intel compiler to gcc.

Then build your MPI program with `mpicc <usual_gcc_flags> -o test_mm test_mm.c`.

Then launch it using TACC's job scheduler SLURM. Normally this is `srun`, but since you are using MPI, you need to use the MPI-aware runner instead `ibrun`. Below is an example of a launch script (write all of the indented text into a file run_mpi.sh) to allocate resources and run `ibrun` with your binary.

```
#!/bin/bash
# filename: run_mpi.sh

#SBATCH -J mpi_mm                           # job name
#SBATCH -o mpi_mm.o%j                       # output and error file name (%j expands to jobID)
#SBATCH -n 256                              # total number of mpi tasks requested
#SBATCH -N 2                                # number of mpi nodes requested
#SBATCH -p normal                           # queue (partition) -- normal, development, etc.
#SBATCH -t 00:00:30                         # run time (hh:mm:ss) - 30 seconds
#SBATCH --mail-user=your.email@utexas.edu
#SBATCH --mail-type=begin                   # email me when the job starts
#SBATCH --mail-type=end                     # email me when the job finishes
ibrun ./mpi_mm 1 0 1024
run code snippet
```

And then you launch with `sbatch run_mpi.sh`.

https://portal.tacc.utexas.edu/user-guides/lonestar6#building-mpi
https://portal.tacc.utexas.edu/user-guides/lonestar6#launching-mpi

Part 2: (20%) Shared Memory in Cilk

Write the same program (with the same Makefile requirements) in Cilk on a single Lonestar6 node (with 2 processors, each with 64 cores for a total of 128 cores.) You will only be responsible for supporting up to 128 cores, running on a single Lonestar6 node. You can only call gen_sub_matrix() once for each submatrix. Since Cilk is shared memory, call gen_sub_matrix once for each matrix to generate the entire matrix. The total memory that you are allowed for the Cilk implementation is (matrix_dimension * matrix_dimension) * (num_matrices + 3).

Use OpenCilk. The pre-build opencilk clang is located at
`/work/08382/mengtian/ls6/cilk/bin/clang`
We recommend you use this pre-build version as it could save compilation time. If you want to try installation on your own here is the guide: Getting started (opencilk.org)
And here is the code.

GitHub - OpenCilk/infrastructure: Installation instructions and build scripts for OpenCilk.

Note that PID is not used in Cilk.
Set the number of workers with `CILK_NWORKERS=N` when running.
You are NOT allowed to use the cilk_for construct.