UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 5/24/2017
Problem Set #8 **Due:** 5/30/2017

1 Preliminaries

1.1 MPI with Docker

Because of the unexpected and last-minute need to set up a full distributed-memory cluster in AWS, we will be doing our preliminary and warm up exercises with MPI in docker. Regarding correctness and concurrency, MPI in docker will work basically like it would in a truly distributed memory system.

(Note, however, that MPI is about communicating sequential *processes* – those processes can actually be on the same node. The issue is whether they can read and write another process's memory. Even on shared memory hardware, separate processes cannot directly read and write each other's memory. So, functionally, running MPI under docker with a few threads but with completely separate processes is just fine for development and testing.)

Rather than using the `amath583/base` image for this assignment, you wil be using the image `amath583/mpich`. This image is the same as the one we have been using so far, but also includes the MPICH implementation of MPI[1]. The steps for launching this image are the same as with the base one. Test out this image by running it and issuing the following command:

```
% mpirun --version
```

You will get back alot of information, but the first line should say "HYDRA build details:" You may also note that the line about `CXX` has `g++` as the associated C++ compiler. We will over-ride this in the Makefile.

1.2 MPI in the Cloud

TBD.

As of now, you should be able to do all of the exercises included in this assignment using docker. Distributed / cloud update will be released shortly.

1.3 Compiling and Running an MPI Program

As we have discussed in lecture on several occasions, MPI programs are not special programs in any way, other than that they make calls to functions in an MPI software library. The MPI program is a plain old sequential program, regardless of the calls to MPI functions. (This is in contrast to, say, an OpenMP program where there are special directives for compilation into a parallel program.)

Given that, we can compile an MPI program into an executable with the same C++ complier that we would use for a sequential program without MPI function calls. However, using a third-party library (not part of the standard library) requires specifying the location of include files, the location of library archives, which archives to link to, and so forth. Since MPI itself only specifies its API, implementations vary in terms of where their headers and archives are kept. Moreover, it is often the case that a parallel programmer may want to experiment with different MPI implementations with the same program.

It is of course possible (though not always straightforward) to determine the various directories and archives needed for compiling an MPI program. Most (if not all) implementations make this process transparent to the user by providing a "wrapper compiler" – a script (or perhaps a C/C++ program) that properly establishes the compilation environment for its associated MPI implementation. Using the wrapper compiler, as user does not have to specify the MPI environment. The wrapper compiler is simply invoked as any other compiler – and the MPI-associated bits are handled automagically.

The wrapper compilers associated with most MPI implementations are named along the lines of "mpicc" of "mpic++" for compiling C and C++, respectively. These take all of the same options as their underlying

[1] I would have obviously preferred to use Open MPI for this course, but for some reason it does not play well with Docker

(wrapped) compiler. The default wrapped compiler is specified when the MPI implementation is built, but it can usually be over-ridden with appropriate command-line and/or environment settings. For MPI assignments in this course, the compiler is `mpic++` – the name used by both MPICH and Open MPI.

1.4 mpirun

Launch the mpich docker image. The first MPI program that you will run is not an MPI program at all. At the command prompt type the following:

```
% hostname
```

As in an early assignment, you will get back the hostname of the current docker container. Now, issue the following command:

```
% mpirun -np 4 hostname
```

The `-np 4` option instructs `mpirun` to launch four processes. This command is a very useful diagnostic for determining whether `mpirun` is functioning (or how it is functioning). It can also be helpful to issue other commands like

```
% mpirun -np 4 pwd
```

What is important in these examples – and why they are diagnostically useful – is that `mpirun` invokes multiple copies (as specified by `-np`) of the indicated program. In a real cluster, these would be on different nodes and you would see different hostnames (although, as with what we are doing here, you may have multiple processes on the same node). In a real cluster, the node where the MPI processes are launched may have a different working directory than the program where you run `mpirun`. (And remember, `mpirun` is only launching your programs – in a real cluster they probably will be run on different nodes than where you build and launch your programs.)

It is also important to note that we are using `mpirun` programs to launch programs that don't have any MPI function calls in them. Again, in some sense, there is no such thing as "an MPI program." There are only processes, some of which may use MPI to communicate with each other. But `mpirun` does not what inspect what the processes actually do, so it will launch programs without MPI calls perfectly fine.

For your first program to try yourself with `mpirun` type in and compile the following C program:

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  (void) printf("pid = %d\n", getpid());
  return 0;
}
```

You can compile by just executing `cc yourprogram.c` – it doesn't matter what you name it, and if you don't specify an output name it will create `a.out`. Now run the program by issuing

```
% mpirun -np 4 ./a.out
```

What gets printed? In all of the previous diagnostic programs, all of the processes responded with the same results, so we just got multiple copies of the same results. Now we finally have processes that give visibly different behavior. (In this example the program prints its own process id).

Experiment a bit with different values after `-np` – and with different programs given to `mpirun`. Note that you can pass arguments to the program you are launching just by tacking them onto the end of the command.

```
% mpirun -np 4 ls -CF
```

will run four copies of `ls -CF`.

1.5 Hello MPI World

For your first non-trivial (or not completely trivial) MPI program, try the hello world program we presented in lecture, provided as `hello.cpp`. The basic command to compile this program is

```
% mpic++ hello.cpp
```

Note however, that if we are going to use all of the other arguments that we have been using in this course with C++ (such as -std=c++11, -Wall, etc), we need to pass those in too. Moreover, we need to make sure we use clang!

What do you get when you execute:

```
% mpic++ --version
```

The released compiler with Ubuntu and `mpich` is g++ – and we want to stick with clang as there are some differences that we might as well avoid. To use clang with `mpic++`, we need to add one more argument: `-cxx=clang++`. To test this, run the following

```
% mpic++ -cxx=clang++ --version
```

You should get a message showing clang 3.8 – the same message as if you just run

```
% clang++ --version
```

Thus, to compile hello.cpp you should issue (at least) the following:

```
% mpic++ -cxx=clang++ hello.cpp
```

which will create an executable a.out. Once you have hello.cpp compiled (into a.out or some other executable), run it!

```
% mpirun -np 4 ./a.out
```

Experiment with different numbers of processes – though you probably don't want to launch more than ten or twelve on docker – since these are real processes that are being launched.

I recommend that you manage the compiler and its flags through Makefile macros. (You can use, modify, or copy from the Makefile provided with this – or any other – assignment). I suggest using

```
CXX                = mpic++ -cxx=clang++
```

to define your compiler – and then building make rules from that macro. If you don't specify your compiler via a macro – just make sure to use `mpic++ -cxx=clang++` rather than `c++`.

1.6 MPI Ping Pong

The point to point example we showed in class used `MPI::Comm::Send()` and `MPI::Comm::Recv()` to communicate – to bounce a message back and forth. Rather than including all of the code for that example in the text here, I will just refer you to the included program `pingpong.cpp`.

Build an executable program from pingpong.cpp (I suggest using pingpong rather than a.out as the executable name). In this case the program can take a number of "rounds" – how many times to bounce the token back and forth. Note also that the token gets incremented on each "volley". Launch this program with `mpirun -np 2`. Note that we are passing command line information in to this program. Note also that we are not testing the rank of the process for the statement

```
  if (argc >= 2) rounds = std::stol(argv[1]);
```

What are we assuming in that case about how arguments get passed? Is that a valid assumption? How would you modify the program to either "do the right thing" or to recognize an error if that assumption were not actually valid?

Does this program still work if we launch it on more than two processes? Why or why not?

2 Warm Up

For this assignment you will be provided with sequential and MPI-parallel versions of the following files: `ir.hpp`, `ir.cpp`, `Grid.hpp`, `Grid.cpp`, `Stencil.hpp`, `Stencil.cpp`, `cg.hpp`, `cg.cpp`, `ir_driver.cpp`, `cg_driver.cpp`, and `Makefile`. You will also be furnished with MPI specific code for hello world and ping pong. These may be incomplete for the cases where you are asked to complete certain parts of the code.

3 Exercises

3.1 Ring

In the MPI ping pong example we sent a token back and forth between two processes, which, while actually quite amazing, is still only limited to two processes.

Write a program `ring.cpp` (perhaps starting with and modifying `pingpong.cpp`) that instead of simply sending a token from process 0 to 1 and then back again, sends a token from process 0 to 1, then from 1 to 2, then from 2 to 3 – etc until it comes back to 0. As with the ping pong example above, increment the value of the token every time it is passed.

Hint: The strategy is that you want to receive from `myrank-1` and send to `myrank+1`. It is important to note that this break down if `myrank` is zero or if it is `mysize-1`. These cases could be addressed explicitly.

Deliverables

- A file `ring.cpp` that implements the aforementioned program.
- A `Makefile` that correctly compiles and creates the program in response to `make ring`.

3.2 mpiTwoNorm

We have had a running example of taking the Euclidean norm of a `Vector` throughout this course. It seems only fitting that an MPI version is created. This has two parts, and the difficult part isn't the one you would expect.

The mpiTwoNorm Function

The statement for this part of the problem is: Write a function `mpiTwoNorm` that takes a `Vector` and returns its Euclidean norm.

That is easy to state, but we have to think for a moment about what it means in the context of CSP. Again, this function is going to run on multiple separate processes. So, first of all, there is not really such a thing as "a Vector" or "its norm". Rather, each process will have a part of a Vector and each process can compute the norm of its part (or it can compute the sum of squares of its part).

But now what do we mean by "its norm"? If we consider the local vectors on each process to be part of a larger global vector, then what we have to do is compute the local sum of squares, add those all up, and take the square root. This immediately raises another question though. We have some number of separate processes running? How (and where) are they "added up"? Where (and how) is the square root taken? Which processes get the result?

Typically, we write individual MPI programs working with local data as if they were a single program working on the global data. That can be useful, but it is paramount to keep in mind what is actually happening.

With this in mind, generally, the local vectors are parts of a larger vector and we do a global reduction of the sum of squares and then either broadcast out the sum of squares (in which case all of the local processes can take the square root), or we reduce the sum of squares to one process, take the square root, and then broadcast the result out. Since the former can be done efficiently with one operation (all-reduce), that is the typical approach.

The precise statement of this part of the problem is: Write a function `mpiTwoNorm` that is executed on each process in a communicator (assume MPI Comm World). The function should take a local `Vector` and return the Euclidean norm of a vector that would be the concatenation of all of the local vectors. All processes calling the function should return the same result.

The mpiTwoNorm Driver

Now here is the harder question. In previous assignments you have been asked to test your implementations of Euclidean norm with a driver. A random vector is usually created to compare the sequential result to the parallel result (or to differently ordered sequential results). But how do we generate the same random vector across multiple independent processes, each of which will be calling its own version of randomize? Consider the following example:

```
int main() {
  MPI::Init();

  size_t myrank = MPI::COMM_WORLD.Get_rank();
  size_t mysize = MPI::COMM_WORLD.Get_size();
  Vector lx(1024);
  randomize(lx);
  double rho = dot(lx, lx);

  std::cout << myrank << ": " << std::sqrt(rho) << std::endl;

  MPI::Finalize();
  return 0;
}
```

Compile this with `mpic++` (per above instructions) and run it. What does it print out?

Here is a situation where the SPMD local processes are quite different from the equivalent global process that we want. By that I mean randomize is a completely sequential process. It uses a pseudo-random number generator that starts with some seed number and then generates a deterministic sequence of (unpredictable) numbers. No matter how many copies of it we run, each copy generates the same sequence – and therefore populates the same values in each local vector. But this is not what we want at all! The local vectors are supposed to be parts of a single longer vector – and, for testing, one that we can replicate sequentially.

One approach to dealing with issues such as this (the same kind of problem shows up when, say, reading data from a file) is to get all the data needed on one node (typically node 0) and then scatter it to the other nodes using `MPI::Comm::Scatter`.

Write a driver program `mpi2norm_driver.cpp` that takes as input the local size of a `Vector`. On rank 0 it should create a `Vector` of that local size times the number of ranks in `MPI::COMM_WORLD`. It should then scatter that vector out to all the other processes – into local vectors of the size given. (Scattering takes one line of code – one invocation of `MPI::Comm::Scatter`). On node 0 the program should compute the Euclidean norm of the global vector and then call your `mpiTwoNorm` function to compute the Euclidean norm across all of the processes. Your program should print the following:

```
#\tglobal\tmpi\tdiff
```

where "backslash t" is the tab character. In the next line, again separated by tabs, you should print the number of processes run, the global norm of the large vector on rank 0, the value returned by `mpiTwoNorm`, and finally, the absolute value of the difference between the two. The print out should only occur on one process (0).

Deliverables

- A file `mpi2norm_driver.cpp` that contains the `mpiTwoNorm` function and a `main` function to drive it, all as described above.

- A `Makefile` that correctly compiles and creates the driver program in response to `make mpi2norm_driver`.

3.3 Timing mpiTwoNorm (AMATH583 only)

In addition to testing, we also have been evaluating our programs for how they scale with respect to problem size and to number of threads/processes. We will continue this process for this problem. In distributed memory, however, the issue of timing becomes a little touchy, because there is no real notion of synchronized time across the different processes – but we need one number to measure to determine whether we are scaling or not.

Make a copy of your `mpi2norm_driver.cpp` and rename it `mpi2norm_timer.cpp`. Add the necessary code to print out the time required to compute the global two norm on rank 0 as well as the time between the scatter and gather for the parallel two norm.

As you have done before, your program should print a line of tab (or space) separated numbers. The first element in the line should be the number of processes, the second should be the size of the vector, the third the sequential execution time, the fourth the parallel execution time, the fifth the speedup, and finally the difference between the computed norms.

Deliverables

- A file `mpi2norm_timer.cpp` that contains the `mpiTwoNorm` function and a `main` function to time it, all as described above.
- A `Makefile` that correctly compiles and creates the driver program in response to `make mpi2norm_timer`.

4 Turning in The Exercises

All your cpp files, hpp files, and your `Makefile` should go in the tarball `ps8.tgz`. If necessary, include a text file ref8.txt that includes a list of references (electronic, written, human) if any were used for this assignment.

Before you upload the `ps8.tgz` file, it is **still very important** that you have confidence in your code passing the automated grading scripts. After testing your code with your own drivers, make use of the python script test_ps8.py by using the command `python test_ps8.py` in your ps8 directory. Once you are convinced your code is exhibiting the correct behavior, upload ps8.tgz to Collect It.

5 Learning Outcomes

At the conclusion of week 9 students will be able to

1. Explain the basic parallelization strategy of MPI
2. Describe communicating sequential processes and SPMD models
3. Name and describe the functions in both the collective and the point-to-point versions of "six-function MPI"
4. Describe the semantics of `MPI::Comm::Send` and `MPI::Comm::Recv`
5. Write "ping pong" programs using "six function MPI" that sends and receives an integer, a double, an array of integers, or an array of doubles
6. Describe what "ghost cells" are, where they arise, and what you do with them
7. Describe a scenario where MPI can deadlock
8. Describe three strategies for avoiding such deadlock

9. Describe the principal components of the MPI communicator abstraction
10. Explain `MPI::Comm::Scatter` and `MPI::Comm::Gather`
11. Explain Cannon's algorithm for distributed matrix-matrix product
12. Describe a strategy and implement a program for computing π using numerical quadrature
13. Describe the semantics of `MPI::Comm::Isend` and `MPI::Comm::Irecv`
14. Describe the semantics of `MPI::Comm::Ssend`