CSE P 524 **Assigned:** 3/10/2022
Problem Set #10 **Due:** 3/18/2022

1 Preliminaries and Warmup, Continued

You should have received preliminaries and warmup for this assignment as starter/ps10/pre-ps10.pdf.

1.1 Compiling and Running an MPI Program

As we have emphasized in lecture, MPI programs are not special programs in any way, other than that they make calls to functions in an MPI software library. An MPI program is a plain old sequential program,[1] 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 (local) 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" or "mpic++" for compiling C and C++, respectively. These take all of the same options as their underlying (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 named "mpic++" (or, equivalently, "mpicxx").

Verify that your "mpicxx" compiler is, in fact, a wrapper compiler.

```
$ mpicxx --version
```

1.2 mpirun

In one of the scripts above, we used "mpirun" to launch a program that didn't have any MPI function calls in it. Again, in some sense, there is no such thing as "an MPI program." There are only local processes, some of which may use MPI library functions to communicate with each other. But "mpirun" does not in any way inspect what the processes actually do, so it will launch programs without MPI just as well as programs that do call MPI.

What it does do is set up a communication runtime system that enables programs that do make MPI function calls to be able to execute those function calls.

1.3 Hello MPI World

To Do 1: For your first non-trivial (or not completely trivial) MPI program, try the hello world program we presented in lecture, provided as "hello.cpp" in your ps10/hello subdirectory.

The basic command to compile this program is

[1] Well.... An MPI program can also be an OpenMP program, or a multithreaded program, or a CUDA program — or all of the above. The point is that it is the same kind of program as you would run locally.

```
$ mpicxx hello.cpp
```

Note however, that 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 – so we rely on "make" to automate this for us.

```
$ make hello.exe
```

which will create an executable hello.exe. Once you have hello.cpp compiled, run it!

```
$ srun --tasks=4 hello.exe
$ srun --nodes=4 hello.exe
$ srun --nodes=4 --tasks=2 hello.exe
```

Reminder: To take full advanatage of the hardware on the compute nodes, you should compile on the compute nodes. E.g.,

```
$ srun make hello.exe
```

There is also a script in your directory that you can experiment with

```
$ sbatch --tasks=4 hello.bash
$ sbatch --nodes=4 hello.bash
$ sbatch --nodes=4 --tasks=2 hello.bash
```

Experiment with different numbers of processes, being aware of the resource limitation of the niac allocation.

In general, you should prefer "sbatch" and scripts to "srun".

Reminder: We have a total of 80 cores in the "niac" partition that can be allocated among an arbitrary number of nodes for students in this course. To avail yourself of nodes in that partition, specify the "niac" account. You can also use the general pool of compute resources – the "ckpt" account. If you don't specify an account, your job will default to niac.

1.4 Ping Pong

The point to point example we showed in class used MPI to communicate – to bounce a message back and forth. Rather than including all of the code for that example in the text here, I refer you to the included program ps10/pingpong.cpp.

To Do 2: Take a few minutes to read through pingpong.cpp and determine how the arguments get passed to it and so forth.

Build the executable program "pingpong.exe". 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

```
$ sbatch --nodes=2 pingpong.bash
```

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? Try launching with

```
$ sbatch --nodes=3 pingpong.bash
```

2 Problems

2.1 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.

To Do 3: Complete the code in "ring.cpp" so that instead of simply sending a token from process 0 to 1 and then back again, the token is sent 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.

Question 1: Include a listing of your `main()` function (or wherever you implement the ring functionality). Provide comments in the code near your edits to explain your approach.

Hint: The strategy is that you want to receive from `myrank-1` and send to `myrank+1`. It is important to note that this will break down if `myrank` is zero or if it is `mysize-1` because there is no rank -1 or mysize. These cases could be addressed explicitly. (Or, since with everything else in MPI, there may be capabilities in the library for handling this situation, since it is not uncommon.)

2.2 Norm

We couldn't visit another high-performance computing paradigm without using it to compute the Euclidean norm of a vector. Computing the Euclidean norm with MPI has two parts, and the difficult part isn't the one you would expect.

2.3 The mpi_norm

The statement for this part of the problem is: Write a function 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/SPMD. 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 its own vector and each process can compute the norm of that vector (or, more usefully, it can compute the sum of squares of that vector because we have to then take a global sum of all of those pieces to compute the final norm). Together, the different vectors on each process can be considered to be one distributed vector – but the distributed vector as an actual "thing" only exists in the mind of the programmer.

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 all of those up, and then take the square root of the resulting global sum. This immediately raises another question though. We have some number of separate processes running. How (and where) are the individual sums "added up"? Where (and how) is the square root taken? Which process gets (or 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.

2.3.1 Distributing the vector

Now here is the harder question that I alluded to above. In previous assignments you have been asked to test and time your implementations of norm. 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 a distributed random vector across multiple independent processes, each of which will be calling its own

version of "randomize()"? More specifically, how do we make a distributed vector that is the "same" as a sequential one (with the same total number of elements)? We need to create the "same" vector if we are going to, say, check it for correctness.

We don't want to have every node separately call "randomize". – in that case, each node would fill in the same values (random numbers on a computer are algorithmically generated – each node runs the same algorithm from the same starting point – hence generating the same values). Instead, we want a single random vector partitioned and distributed across the nodes.

To properly compute the norm of a vector in parallel, first, let's get a truly distributed random vector (so that each process has a piece of a larger vector, not the same copy of a smaller vector). 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_Scatter."

Second, we need to get the final computed norm to all of the processes rather than just to a single process.

Finally, for testing, we want to check that all of the compute nodes have the same value. We can make this check on one node, using, say, the inverse operation of scatter (gather).

Deliverable 1: Complete the program "mpi_norm".

1. The program takes in the (global) size of a vector (log_2 of its size). It should create a random vector of the indicated size and then scatter the vector out to all of the other processes (assume everything is a power of two) using an MPI collective operation.

2. The compute nodes should compute their local contributions to the global norm. Use another MPI collective operation to combine those results and send the results to all of the compute nodes so that a consistent value of the norm can be computed on all compute nodes. To realize this, complete the function "mpi_norm" in "mpi_norm.cpp".

In addition to testing for correctness, 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 time across the different processes – but we need one number to measure to determine whether we are scaling or not.

Question 2: What changes did you make for `mpi_norm`? Include a listing of your `mpi_norm` function. Provide comments in the code near your edits to explain your approach.

To Do 4: Run the script "strong.bash". This will run "mpi_norm.exe" for a set of problem sizes for different numbers of processes and plot the resulting times.

Question 3: Include the generated pdf file.

To Do 5: Run the script "weak.bash". This will run "mpi_norm.exe" for different sizes and different numbers of processors – keeping the local problem size constant.

Question 4: Include the generated pdf file.

Question 5: Per our discussions in lectures past about weak vs strong scaling, do the plots look like what you would expect? Describe any (significant) differences from your expectations (if any).

Question 6: For strong scaling, at what problem size (and what number of nodes) does parallelization stop being useful? Explain.

2.4 N body

The N-body problem became computationally expensive with problem size very quickly since the computation is $O(N^2)$. That is good news and bad news for parallelization. The good news first. Each body requires information from every other body. If we were to partition the bodies, we still need information from all of the others in the "ghost" bodies so we might as well keep all of the bodies on each compute node

and have each compute node only responsible for updating part of the system. Keeping the entire system can be bad if it is a large system. And we could develop a more sophisticated scheme where parts of the system are circulated through the compute nodes. However, since there is so much computing to do relative to the data size, keeping all of the data will still show us good speedup for the class of problems we are going to look at. The bad news, of course, is that we can't really scale the problem indefinitely. If we grow the problem by a factor of 10, the work grows by a factor of 100. There are more sophisticated algorithms that have $N\log N$ complexity or even $O(N)$ complexity (Barnes-Hut and Fast Multipole, respectively), but those are also beyond the scope of this assignment.

2.4.1 Initialization

Since this is an SPMD program, we are going to be timestepping, computing (a subset of) body interactions, etc on every node. Accordingly, we need all of the information about the simulation on every node. As with our other programs, we let rank 0 read in the command-line arguments and broadcast those values to the other nodes.

```
MPI::COMM_WORLD.Bcast(&dt, 1, MPI::FLOAT, 0);
MPI::COMM_WORLD.Bcast(&num_timesteps, 1, MPI::UNSIGNED_LONG, 0);
MPI::COMM_WORLD.Bcast(&num_bodies, 1, MPI::UNSIGNED_LONG, 0);
```

Next, we create a vector for all of the bodies on every node. However, we randomize them on rank 0 and then broadcast them to every other node so that all nodes have the same initial conditions.

```
std::vector<body<float>> bodies(num_bodies);
if (0 == myrank) {
  randomize(bodies);
}
MPI::COMM_WORLD.Bcast(bodies.data(), (int)bodies.size(), make_body_type(), 0);
```

2.4.2 Computation

Then, as with the sequential and OpenMP versions of the program, we invoke `run` (which looks just like our previous versions):

```
template <typename T, typename TimeType>
void run(std::vector<body<T>>& bodies, TimeType dt, TimeType end_time) {
  accumulate_fields(bodies);
  offset_velocities(bodies, -dt / 2);

  for (TimeType t = 0; t < end_time - dt / 2; t += dt) {
    update_state(bodies, dt);
    accumulate_fields(bodies);
  }
}
```

And, in fact, `update_state` looks just the same as before:

```
template <typename T, typename TimeType>
void update_state(std::vector<body<T>>& bodies, TimeType dt) {
  for (auto& b : bodies) {
    b.vel += (G * dt) * b.field;
    b.pos += (dt * b.vel);
  }
}
```

However, `accumulate_fields` will look quite a bit different. When we parallelized nbody with OpenMP, we parallelied the loops in `for_each_pair`. In our SPMD approach, we will be partitioning the problem

in somewhat the same way – each process will operate on a portion of the bodies (and this partitioning will be the same as we did for OpenMP in order to get proper load balancing). However, we also need to communicate the results of those updates to all of the other processes.

To do this, we create our partitions and save the results in a vector `lb`. Each process then determines its own local end and begin, based on its MPI rank (its `id`) and then each process invokes `for_each_pair` on its local part of the problem.

```
size_t id          = MPI::COMM_WORLD.Get_rank();
size_t num_threads = MPI::COMM_WORLD.Get_size();

static std::vector<double> lb = make_lbs(num_threads);

auto   begin = bodies.begin();
auto   end   = bodies.end();
size_t len   = end - begin;

auto local_begin = begin + len * lb[id];
auto local_end   = begin + len * lb[id + 1];

for_each_pair(local_begin, local_end, [](body<T>& pi, body<T>& pj) {
  auto u = ror3(pj.pos, pi.pos, 1.e-3f);
  pi.field += pj.mass * u;
  pj.field -= pi.mass * u;
});
```

Then, each process needs to send its contribution to every other process (more specifically they ALL need to be GATHERED). To do this, we need to define an MPI datatype to represent a body, which we do in `make_body_type` (refer to the code for details). We also need to set up information about the communication operation: The number of elements that are received from each process (`recvcounts`) and the location (relative to the receive buffer) at which to place the incoming data from each process (`displs`).

```
static auto body_type = make_body_type();

std::vector<int> recvcounts(num_threads);
std::vector<int> displs(num_threads);
for (size_t i = 0; i < recvcounts.size(); ++i) {
  recvcounts[i] = len * (lb(i+1, num_threads) - lb(i, num_threads));
}
std::partial_sum(recvcounts.begin(), recvcounts.end(), displs.begin()+1);
```

Finally, we need to GATHER the information we need from ALL the processes.

Deliverable 2: Complete the implementation of `accumulate_fields`. This should only require one MPI function call (and there are several hints above as to what that should be and what the arguments to it should be).

The arguments to the MPI version of nbody are similar to what they were for the OpenMP case. To run nbody.exe on multiple nodes, use srun (or sbatch).

```
$ srun --nodes=4 ./nbody.exe -t 1024 -n 1024
```

This will run nbody.exe on 4 separate nodes, with 1024 bodies and 1024 timesteps.

Question 7: How does nbody.exe behave for the following cases:

```
$ srun --nodes=4 --tasks-per-node=1 ./nbody.exe -t 1024 -n 2048
$ srun --nodes=8 --tasks-per-node=1 ./nbody.exe -t 1024 -n 2048
$ srun --nodes=4 --tasks-per-node=2 ./nbody.exe -t 1024 -n 2048
$ srun --nodes=2 --tasks-per-node=4 ./nbody.exe -t 1024 -n 2048
$ srun --nodes=1 --tasks-per-node=8 ./nbody.exe -t 1024 -n 2048
```

Question 8: Your starter code includes a bash script strong_script.bash that you can use to run scaling studies on small, medium, and large problem sizes. The script will produce a PDF file strong.pdf. Include strong.pdf in your writeup for this question.

Question 9: `srun` will also take an argument `--cpus-per-task`. How does nbody.exe behave for the following cases:

```
$ srun --nodes=4 --cpus-per-task=1 ./nbody.exe -t 1024 -n 2048
$ srun --nodes=4 --cpus-per-task=2 ./nbody.exe -t 1024 -n 2048
$ srun --nodes=4 --cpus-per-task=4 ./nbody.exe -t 1024 -n 2048
```

Explain. (What is the difference between nodes, tasks-per-node, and cpus-per-task? How can each contribute to parallelization or not? What does your program have to do to take advantage of each or not?)

Question 10: Now that you have vectorization, multithreading, OpenMP, CUDA, and MPI in your repertoire, suggest a strategy for most effectively using a cluster of modern multicore CPUs. A cluster of modern multicore CPUs with GPUs? What are some advantages to your strategy? (Under what conditions will it be most effective?) What are some disadvantages to your strategy? (Under what conditions will it be least effective?)

Deliverable 3: (Extra Credit) Incorporate some mechanism(s) into your nbody code so that running with more CPUs per task will provide some speedup.

Question 11: (Extra Credit) Include a listing of your changed code.

3 What to Turn In

All of the code for this assignment should be checked into your course shared repository. You should also check in norm/strong.pdf, norm/weak.pdf, and nbody/strong.pdf. Your repository should also include the PDF of your written answers, saved as "ps10-written.pdf." Tag your final commit as "ps10_submission".

Checklist:

- Append a list of sources / references you used for this assignment to the end of ps10-written.pdf.
- Upload your ps10-written.pdf document to GradeScope under assignment "ps10".
- Upload your `ring.cpp`, `mpi_norm.cpp`, and `nbody.cpp` to GradeScope under assignment "ps10_code".