CSc 522, Program #1: Shared- and Distributed-Memory Parallel Scientific Programming

Due date: Thursday, October 3rd, 2019, at 8am. **No late assignments will be accepted.**

This assignment has two purposes: (1) to introduce you to shared, and distributed, and hybrid parallel programming, and (2) to help you gain experience with computer science systems experimentation. This assignment must be done in C or C++.

Start from the sequential Jacobi iteration program located at: `http://dkl.cs.arizona.edu/teaching/csc522-fall19/assignments/seq-jacobi.c`. You are required to do the following, using the cluster of multicore machines (running OSX) on the 9th floor.

1 Multithreaded Program

- Write a multithreaded versions of Jacobi iteration. Use `pthreads` for the shared-memory parallel program. You must write a coarse-grain program (1 thread/core, assuming a small number of cores). So, as with the matrix multiplicaton example in class, each thread should work on a strip of rows.
- Use a dissemination barrier for synchronization. You will likely need to use the `volatile` qualifier on the `arrive` array in the solution provided in class.
- Please think carefully about where to insert barriers. Be sure to document—in the code—why you inserted each barrier. Note that each iteration does two sweeps; you cannot change that.
- You may not use locks in this program. Instead, you can use a global array to hold the per-thread maximum differences; then, one thread can find the largest difference.
- Perform speedup measurements on 2 and 4 cores. Note that I reserve the right to ask for speedup on 8 cores also, but I am not doing so yet. (This is nearly zero extra work, if I do ask.) Your experiments must be carefully done; for example, you need to coordinate with each other to find times that both (1) your classmates are not running tests, and (2) others are not running CPU-intensive jobs. Note that speedup is computed relative to the *sequential* program, not the one-thread parallel program.

2 Message-Passing Program

- Write a message-passing version of the program using MPI. For this part, you should have each process initialize its own subarray (this means you do not have to explicitly disseminate the data as was done in the matrix multiplication example). Use `MPI_Allreduce` to determine the final maximum difference; do not use a separate coordinator process. Use `MPI_Wtime` for timing; note that the program is done when all processes are done, so only a single (whole program) time (not one per process) must be used.

  You will need to do message exchanges. Note that if you have all MPI processes perform `MPI_Send` followed by `MPI_Recv`, the program will deadlock if the messages is sufficiently large. Instead, the pattern you must use is `MPI_Irecv`, followed by `MPI_Send`, followed by an `MPI_Wait` (where the `MPI_Wait` matches the `MPI_Irecv`).

- As with the multithreaded version, note that each iteration does two sweeps, and you cannot change that. You should think carefully when and where to do message exchanges.
- Perform speedup measurements of your MPI program on 2, 4, and 8 nodes. Note that speedup is computed relative to the *sequential* program, not the one-process parallel program.
- Note that to run an MPI program over different nodes, you must be able to ssh in to each of those nodes without a password. If you do not already have this capability, see for example `http://www.linuxproblem.org/art_9.html`.

3 Hybrid Program

- Finally, write a *hybrid* program that simultaneously exploits shared- and distributed-memory parallelism. For this, use pthreads within the processes and MPI between processes. If you execute your program on N nodes, where each node has P cores, then you must have N MPI processes, where each process uses P threads. Use `MPI_THREAD_FUNNELED` as the flag to `MPI_Init_thread`. That is, only the main thread (the one that calls `MPI_Init_thread`) makes MPI calls. As with the message-passing program, you should have each process initialize its own subarray.
- Unlike the multithreaded program, you should not use a dissemination barrier. Instead, on each iteration, you must create and then join the threads that will do the work. Please think carefully about this. (The join is serving as the barrier.) This means that you will need to split the two sweeps into two separate functions and actually create and join threads twice in each iteration. The reason for this is that it makes it easier to ensure that *only the main thread calls MPI routines*.
- You must use a shared memory array (indexed by thread) to find the maximum difference per MPI process, and then you must use `MPI_Allreduce` to determine the final maximum difference (across all processes).
- I strongly suggest that you make sure both your multithreaded and message-passing versions work before attempting the hybrid program. If you are careful, then the hybrid program is (mostly) a logical composition of the multithreaded and message-passing programs.
- Perform speedup measurements of your MPI program on 2, 4, and 8 processes, with 2 and 4 cores per process. (This means I want you to execute the program 6 times.) Note that speedup is computed relative to the *sequential* program, not the one-process, one-thread parallel program.

Assumptions

You may make the following assumptions:

- The grid size is square, with the dimension a multiple of 64. If the grid size is n, that means that you must allocate a grid of $(n + 2) \times (n + 2)$.
- For the sequential program, the user enters two command line arguments: gridsize and number of iterations, in that order.

- For the multithreaded program, the user enters three command line arguments: gridsize, number of iterations, and number of threads, in that order.
- For the hybrid program, the number of processes is again passed as a parameter to `mpirun`, so the command line arguments are, as for the multithreaded program, the gridsize, number of iterations, and number of threads, again in that order.
- The user will never make an input error.

Output

You must output the following items, to stdout, in the exact order listed, delimited by a single space:

- number of MPI processes (0 for the pthreads and sequential programs),
- number of threads (0 for the sequential and pure MPI programs),
- execution time, in seconds, to three places to the right of the decimal point, and
- the maximum difference on the final iteration, to five places to the right of the decimal point.

Use `gettimeofday` (for the sequential and pthreads programs) and `MPI_Wtime` (for the MPI and hybrid programs). For the multithreaded program, start the clock just before you create threads; this must be after the initialization phase, which should be sequential. Stop the clock after a barrier at the end of the code. For the MPI and hybrid program, start the clock after the initialization phase and stop the clock right before `MPI_Finalize`.

Submitting the Assignment

You must submit your assignment using the `turnin` command on `lectura` as follows:

```
turnin csc522-f19-prog1 prog1.tar
```

where the tar file includes all of your files. With your code, you must submit a makefile, for which the commands *make seq-jacobi*, *make mt-jacobi*, *make dist-jacobi*, and *make hybrid-jacobi* correctly create executables *seq-jacobi* (the sequential version), *mt-jacobi* (the multithreaded version), *dist-jacobi* (the MPI version), and *hybrid-jacobi* (the hybrid version), respectively. You must compile with optimization level 2, i.e. `gcc -O2` for the sequential and multithreaded programs, and `mpicc -O2` for the MPI program. Along with your programs, you must submit experiments with two different grid sizes, both with a number of iterations equal to 30. The first grid size should be such that the sequential program takes less than 0.5 seconds, and the second grid size should be such that the sequential program takes around 30 seconds. Reported times for each test should be the median of three runs. The results you collect should be presented clearly in a table or graph (add this as a file in your submission, and call it `results.pdf`).

Grading

Grading will be based on several items: (1) correctness, (2) efficiency, (3) code quality, and (4) following directions. Note that *achieving speedup* is not on the preceding list. Certainly, if your code is correct but terribly inefficient in parallel (e.g., you create only one thread on four cores, so get zero speedup), you will be penalized heavily. But please do not operate under the belief that class grades are sorted by speedup.