CSCE 735 Fall 2021

Major Project

Due: 11:59pm Wednesday, December 15, 2021

(No late submission allowed)

This project consists of developing an MPI-based quicksort code and studying its performance. Part A consists of parallelizing a quicksort code by inserting MPI calls. Part B consists of studying its performance in terms of the scalability of the code.

Part A: MPI-based Quicksort on Hypercube

You are provided with a program `qsort_hypercube.cpp` that implements the parallel quicksort algorithm on a hypercube using MPI. At the start of the algorithm, an unsorted list of elements is distributed equally across all processes. At the end of the algorithm, the sorted list is distributed across processors in order of their rank, i.e., the elements on processor P_j are no larger than elements on P_k, for $k > j$. In addition, the elements on a processor are themselves in a sorted list. It is not necessary for the sorted list to be distributed equally across the processors.

The `main` routine is missing **nine** calls to MPI routines that are labeled MPI-1, MPI-2, …, MPI-9. You are required to add these calls to the code to obtain a functioning program. The location of the missing MPI calls is marked by the following text:

```
// ***** Add MPI call here *****
```

The text is preceded by comments that describe what the call should do. When adding the MPI calls to your code, keep in mind:

- IDs of neighbor processes for sends and receives are their IDs in `MPI_COMM_WORLD`.
- Reduction and/or broadcast of the pivot among processes is restricted to each sub-hypercube, and therefore, should use the communicator of the sub-hypercube that a process is a member of.
- MPI Send and Receive calls need pointers to the arrays that are being communicated.

Once you complete the code, it can be compiled with the command:

```
mpiicpc -o qsort_hypercube.exe qsort_hypercube.cpp
```

Setting VERBOSE to a value of 2 or 3 will result in additional output that could help in code development. To execute the program, use

```
mpirun –np <p> ./qsort_hypercube.exe <n> <type>
```

where <p> is the number of MPI processes, <n> is the size of the local list of each MPI process, and <type> is the method used to initialize the local list. The output of a sample run is shown below.

```
mpirun –np 8 ./qsort_hypercube.exe 6000000 0
```

```
[Proc: 0] number of processes = 8, initial local list size = 6000000,
hypercube quicksort time = 0.780582

[Proc: 0] Congratulations. The list has been sorted correctly.
```

1. (60 points) Complete the MPI-based code provided in `qsort_hypercube.cpp` to implement the parallel quicksort algorithm for a d-dimensional hypercube with p=2^d processors. 40 points will be awarded if the code compiles and executes the following command successfully.

   ```
   mpirun -np  2 ./qsort_hypercube.exe 4 -1
   ```

   5 points will be awarded for each of the following tests that are executed successfully.

   ```
   mpirun -np  4 ./qsort_hypercube.exe 4 -2
   mpirun -np  8 ./qsort_hypercube.exe 4 -1
   mpirun -np  16 ./qsort_hypercube.exe 4 0
   mpirun -np  16 ./qsort_hypercube.exe 20480000 0
   ```

Part B: Scalability Study

Weak Scalability Study

2. (5 points) In your own words, describe what is meant by *weak scalability*, i.e., what does it mean that an algorithm or implementation is "weakly" scalable. If you use a textbooks or an online resource as reference, then include a citation.

3. (15 points) Run your code to sort a distributed list of size $n \times p$ where n is the size of the local list on each process and p is the number of processes. For your experiments, use n=20,480,000 and p = 1, 2, 4, 8, 16, 32, and 64. Set type=0. Plot the execution time, speedup, and efficiency of your code as a function of p. Use logarithmic scale for the x-axis.

   Note that the size of the list to be sorted is proportional to the number of processes p. In order to get speedup for a specific value of p, you need to determine the execution time to sort a list of size $n \times p$ with **one** process. As an example, speedup for p = 4 is the ratio of execution time for a list of size 81,920,000 with one process (T_1) to the execution time for a list of size 20,480,000 with 4 processes (T_4).

Strong Scalability Study

4. (5 points) In your own words, describe what is meant by *strong scalability*, i.e., what does it mean that an algorithm or implementation is "strongly" scalable. If you use a textbooks or an online resource as reference, then Include citations as appropriate.

5. (15 points) Now run your code with n=20,480,000/p where p = 1, 2, 4, 8, 16, 32, and 64. Set type=0. Plot the execution time, speedup, and efficiency of your code as a function of p. Use logarithmic scale for the x-axis.

   Unlike the weak scalability study, here the size of the list to be sorted remains unchanged at 20,480,000 even as you increase the number of processes. To determine speedup for any p you need to compare the execution time on p processes to the execution time for a list of size 20,480,000 with one process.

Submission: You need to upload the following to Canvas as a **single zip file**:

1. Part A: Submit the file `qsort_hypercube.cpp`.
2. Part B: Submit a single PDF or MSWord document with your responses to Problems 2-5.

Helpful Information:

1. You should use Grace for this assignment.
2. Load the intel software stack prior to compiling your program. Use:
   ```
   module load intel
   ```
3. Compile MPI programs using `mpiicpc`. For example, to compile `code.cpp` to create the executable `code.exe`, use
   ```
   mpiicpc –o code.exe code.cpp
   ```
4. The run time of a code should be measured when it is executed in dedicated mode. Create a batch file and submit your code for execution via the batch system on the system.