SIENAcollege
Computer Science

Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2024

Programming Project 4: Collective Communication

Due: 4:00 PM, Tuesday, October 8, 2024

In this programming project, you will work with more of MPI's collective communication functionality.

You may work alone or with a partner on this programming project.

There is a significant amount of work to be done here. It will be difficult if not impossible to complete the assignment if you wait until the last minute. A slow and steady approach will be much more effective.

Learning goals:

1. To learn about more efficient communication patterns.
2. To gain experience using MPI collective communication functionality.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `coll2-proj-yourgitname`, for this programming project. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

You may choose to answer the lab questions in the `README.md` file in the top-level directory of your repository, or upload a document with your responses to your repository, or add a link to a shared document containing your responses to the `README.md` file.

Improving RDS

Section 3.4.1 of Pacheco shows how we can improve the gathering of values from the processes at all ranks back to the rank 0 process for operations such as a global sum. Such a mechanism can be used to improve the speed of the communication at end of the RDS program from the earlier lab.

Practice Program: Copy your RDS program to the `tree` directory of your repository and rename it as `mpirds-tree.c`. Replace the $(p-1)$-step computation of the global sums with one using a tree-structured global sum, but using only point-to-point communication. You may use a structure like that in either Figure 3.6 or Figure 3.7, or one of your own design, as long as it has the same efficiency. You may assume that the total number of processes is a power of 2 (but do error checking on this if you make this assumption). No collective communication functions yet! (10 points)

For the questions below, assume you are running your original or improved programs with 512 processes. You don't need to run it, just base your answers on your code.

Question 1: How many total messages are sent and received in your original program to compute the global sum? (1 point)

Question 2: How many "communication phases" are needed by your original program to compute the global sum? That is, what is the longest sequence of consecutive sends or receives by any one process? (1 point)

Question 3: How many addition operations are needed by your original program in the computation of the global sum? (1 point)

Question 4: How many total messages are sent and received in your improved program to compute the global sum? (1 point)

Question 5: How many "communication phases" are needed by your improved program to compute the global sum? That is, what is the longest sequence of consecutive sends or receives by any one process? (1 point)

Question 6: How many addition operations are needed by your improved program in the computation of the global sum? (1 point)

Of course you have been doing the readings and keeping up with your lab work, so you have realized that this communication can be done with one of MPI's higher-level collective communication routines: an `MPI_Reduce`.

Practice Program: Copy your RDS program to the `reduce` directory of your repository and rename it as `mpirds-reduce.c`, replace the computation of the global sums with an appropriate function call to do the entire reduction. (5 points)

A Monte Carlo Method to Compute π

There is a class of algorithms knows as *Monte Carlo methods* that use random numbers to help compute some result.

We will write a parallel program that uses a Monte Carlo method to estimate the value of π.

The algorithm is fairly straightforward. We repeatedly choose (x, y) coordinate pairs, where the x and y values are in the range 0-1 (*i.e.*the square with corners at $(0, 0)$ and $(1, 1)$. For each pair, we determine if its distance from $(0, 0)$ is less than or equal to 1. If it is, it means that point lies within the first quardant of a unit circle. Otherwise, it lies outside. If we have a truly random sample of points, there should be an equal probability that they have been chosen at any location in our square domain. The space within the circle occupies $\frac{\pi}{4}$ of the square of area 1.

So we can approximate π by taking the number of random points found to be within the unit circle, dividing that by the total number of points and multiplying it by 4!

A sequential Java program that uses this method to approximate π is included for your reference in the `pi` directory of your repository.

Practice Program: Write a C program `pi.c` in the `pi` directory of your repository. Your program should be parallelized with MPI, and compute an approximation of π using the Monte Carlo method described. See below for some additional requirements and suggestions. (25 points)

- Your program should take a single command-line parameter, which is the number of random points to generate *on each process*. Store this in a `long` so you can generate large numbers of points to get good approximations. Convert this to a `long` only on the rank 0 process (with good error checking) and use MPI to broadcast the value to all other processes. If the rank 0 process finds an error condition when parsing the command-line parameter, it should call `MPI_Abort` to terminate the computation.
- Use the `drand48` function to generate your random numbers. Each process needs to seed the random number generator with a different value so they all will compute a different pseudorandom sequence. You might make the seed a function of the current time, the rank, and maybe the number of processes.
- No process other than the rank 0 process should produce output.
- After each process has generated its random points and counted the number that lie within the unit circle, gather all of those counts back to the rank 0 process so it can print out information and compute the approximation of π.

Here is a sample run of my program, on 4 processes with 100,000,000 points per process. Your program should output the same information in a similar format. Of course, we are choosing random numbers, so your answers will vary.

```
Will use 100000000 points per process
[0] 78540219 in circle, pi approx = 3.141609
[1] 78538052 in circle, pi approx = 3.141522
[2] 78541818 in circle, pi approx = 3.141673
[3] 78543977 in circle, pi approx = 3.141759
in circle values range from 78538052 to 78543977
Final approximation of pi: 3.141641
```

Question 7: Run your program for 1 billion points per process on 48, 96, and 192 processes on `Stampede3`. Rename your output files to `stampede48.output`, `stampede96.output`, and `stampede192.output`, and include them in your repository. (4 points)

Submission

Commit and push! Make sure your answers to lab questions are provided using one of the mechanisms mentioned in the "Getting Set Up" part of the lab.

Grading

This assignment will be graded out of 50 points.

Feature	Value	Score
`mpirds-tree.c`	10	
Question 1	1	
Question 2	1	
Question 3	1	
Question 4	1	
Question 5	1	
Question 6	1	
`mpirds-reduce.c`	5	
`pi.c` command-line parameter handling/checking/broadcast	5	
`pi.c` random numbers	3	
`pi.c` each rank computes its count	6	
`pi.c` gather counts to rank 0	6	
`pi.c` print counts/pi approximations	5	
Question 7: output files	4	
Total	50	