Prof. Jingke Li (lij@pdx.edu); Classes/Labs: TR 1400-1550 @FAB 150, 88-10; Office Hours: TR 1300-1400 @FAB 120-06

Lab 7: Programming with MPI

1 Simple Send-Receive

Read and understand the program `simple.c`. It implements a pair of send-receive actions between two processes. The sender sends an integer to the receiver; the receiver decreases the value by one and sends it back. Even though the message-passing happens only between two processes, the program can run with any number of processes. It can also take a command-line argument, an integer to be used as the message value. Compile and run it. Here are some examples:

```
linux> mpicc -o simple simple.c     // compile
linux> mpirun -n 2 simple           // run with 2 procs
linux> mpirun -n 4 simple           // run with 4 procs
linux> mpirun -n 4 simple 100       // run with 4 procs, msg=100
```

Your Task Write a program `ring.c` based on the above program. Instead of send-receive actions between two processes, the new program will involve *all* active processes. In the program, process 0 sends an integer to process 1; upon receiving the integer, process 1 decreases its value by 1, and sends the new number to process 2; process 2 does the same, and sends a new number to process 3; and so on. The last process in the active set sends its modified number back to process 0. Like in `simple.c`, each process should make the sending and receiving actions visible by printing out a message showing its rank, its host name, and the involved integer's value. Note that the total number of active processes is not controlled by the MPI program itself. Compile your program with `mpicc`, and test it with multiple combinations of runtime parameters.

2 Block Partition

Read and understand the program `block.c`. It uses a simple division operation to partition an interval range `[0..N-1]` evenly into `P` (#processes) blocks. In the program, process 0 reads in the parameter `N` and broadcasts it to all other processes. Each process computes the boundaries of its corresponding block based on `N` and its own `rank`. All processes then report their block size to process 0 (through the `MPI_Reduce()` collective communication routine), which prints out the sum result. If `P` evenly divides `N`, every value in the range is included in a block, and the total block-size sum equals to `N`. Otherwise, some values will not be in any block, and the block-size sum will be less than `N`.

Your Task Write a second version of this program in `block2.c`. In this version, the blocks cover all values in the range `[0..N-1]`, and their size difference is at most 1. Here is a sample output from this version:

```
linux> mpirun -n 4 block2
Enter the array size: 50
p1 block = [13,25] (size=13)
p2 block = [26,37] (size=12)
p3 block = [38,49] (size=12)
p0 block = [0,12] (size=13)
Total items = 50 (N = 50)
```

Find a simple way for each process to calculate their block's boundaries, and size.

3 Distributing Array

The program `scatter.c` follows up on `block.c`'s partition work, and adds the task of distributing a data array (through the `MPI_Scatter()` routine). Process 0 initializes an array `data[N]`, and scatter it to all other processes

(each process receives a different array block). The processes each computes the sum of the array elements in their block, and then collectively compute the global sum through the `MPI_Reduce()` routine. Read and understand the program. Note that this program uses the simple partition method, and it would not work when `P` does not evenly divide `N`.

Your Task [*Challenging*] Write a second version of this program in `scatter2.c`, to handle the general case. For this program, you'll need to use the variable-sized scatter routine, which has the following interface:

```
MPI_Scatterv(data, cnt, disp, MPI_INT, rbuf, blocksize, MPI_INT, 0, MPI_COMM_WORLD);
```

- `data` – the source array, of size `P`
- `cnt` – an array of size `P`, holding block sizes
- `disp` – an array of size `P`, holding starting index of each block
- `rbuf` – the destination array, of size `blocksize`
- `blocksize` – block size (*Note:* each process can use its own `blocksize`)

The new work for this program all happen to process 0. In addition to initializing the source data array, it needs to prepare two additional arrays, `cnt[P]` and `disp[P]`. As an example, assume `N=50` and `P=4` (see the example in Section 2). We'll have

```
cnt[p] = {13,13,12,12}   // block size for each process
disp[p] = {0,13,26,38}   // starting idx for each block
```

There is a connection between these arrays:

```
disp[k] = cnt[k-1] + disp[k-1];
```

A sample run of this program:

```
linx> mpirun -n 4 scatter2
Enter the array size: 50
p0 block = [0,12] (size=13), rbuf = [1,2,...]
p1 block = [13,25] (size=13), rbuf = [14,15,...]
p2 block = [26,37] (size=12), rbuf = [27,28,...]
p3 block = [38,49] (size=12), rbuf = [39,40,...]
Sum of data = 1275 (ref:1275)
```

Submission

As usual, write a short report summarizing your work. Submit it with your programs, `ring.c`, `block2.c`, and `scatter2.c`, through the "Lab 7" folder on Canvas. The submission deadline is the end of tomorrow (Friday).