Prof. Jingke Li (FAB 120-06, lij@pdx.edu); Class: TR 16:40-17:55 @ ASRC 230; Lab: F 10:30-11:50 @ FAB 88-10.

Lab 8: Sorting and MPI

In this lab, you are going to implement several versions of two sorting algorithms, and practice with a few MPI collective routines. Download and unzip the file `lab8.zip` from D2L. You'll see a `lab8` directory with some program files.

1. Odd-Even Sort

1. Review the odd-even sort algorithm from the lecture notes.
2. The file `oddeven0.c` contains some starter code. Copy it to `oddeven1.c` and complete the implementation. Follow the requirements specified in the comment blocks. You should try to match the sample output shown here:

```
linux> ./oddeven1
[Init] 7 10 5 8 2 15 3 1 14 4 16 13 9 6 11 12
[t= 1] 7 10 5 8 2 15 1 3 4 14 13 16 6 9 11 12
[t= 2] 7 5 10 2 8 1 15 3 4 13 14 6 16 9 11 12
[t= 3] 5 7 2 10 1 8 3 15 4 13 6 14 9 16 11 12
[t= 4] 5 2 7 1 10 3 8 4 15 6 13 9 14 11 16 12
[t= 5] 2 5 1 7 3 10 4 8 6 15 9 13 11 14 12 16
[t= 6] 2 1 5 3 7 4 10 6 8 9 15 11 13 12 14 16
[t= 7] 1 2 3 5 4 7 6 10 8 9 11 15 12 13 14 16
[t= 8] 1 2 3 4 5 6 7 8 10 9 11 12 15 13 14 16
[t= 9] 1 2 3 4 5 6 7 8 9 10 11 12 13 15 14 16
[t=10] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[t=11] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[t=12] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[t=13] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[t=14] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[t=15] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[t=16] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Result verified!
```

3. Compile your program and run several times. Do you observe the array being sorted before all iterations are executed (as shown in the above example)?
4. Now improve your code by adding an early termination detection in your program. The new program should terminate as soon as the array is sorted. Here is a sample output from the new version:

```
linux> ./oddeven2
[Init] 7 10 5 8 2 15 3 1 14 4 16 13 9 6 11 12
[t= 1] 7 10 5 8 2 15 1 3 4 14 13 16 6 9 11 12
[t= 2] 7 5 10 2 8 1 15 3 4 13 14 6 16 9 11 12
[t= 3] 5 7 2 10 1 8 3 15 4 13 6 14 9 16 11 12
[t= 4] 5 2 7 1 10 3 8 4 15 6 13 9 14 11 16 12
[t= 5] 2 5 1 7 3 10 4 8 6 15 9 13 11 14 12 16
[t= 6] 2 1 5 3 7 4 10 6 8 9 15 11 13 12 14 16
[t= 7] 1 2 3 5 4 7 6 10 8 9 11 15 12 13 14 16
[t= 8] 1 2 3 4 5 6 7 8 10 9 11 12 15 13 14 16
[t= 9] 1 2 3 4 5 6 7 8 9 10 11 12 13 15 14 16
[t=10] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[t=11] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[t=12] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Array sorted in 12 iterations
Result verified!
```

2. Bucket Sort

The file `bsort1.c` constains an implementation of bucketsort. It takes one or two command-line arguments:

```
linux> ./bsort1 B [N]   -- use B buckets to sort N numbers (N defaults to 16)
```

It assumes `B` is a power of 2 (so that bucket distribution can be based the leading bits).

1. Read and understand the program. Pay special attention to the routine for deciding which bucket an array element should be placed in:

   ```
   // Find bucket idx for an int value x of b bits
   //
   int bktidx(int b, int x, int B) {
     return x >> (b - (int)log2(B));
   }
   ```

   In the program, the data value range is set to 13 bits, or [0, 8191]. If we use 4 buckets, then the leading 2 bits are used as the bucket index. For example, for the number 4734 (=1001001111110), the leading 2 bits are 10, so it is to be placed in bucket[2].

2. Compile and run this program with different parameter combinations.

3. Write a second version, `bsort2.c`. It is identical to `bsor1.c` except that it reads input from a file, and output result to a file:

   ```
   linux> ./bsort2 B <infile> <outfile>
   ```

   The array size `N` is to be derived from the input file size:

   ```
   FILE *fin = fopen(argv[2], "r");  // open file for reading
   fseek(fin, 0L, SEEK_END);         // go to the end of file
   int size = ftell(fin);            // this gives the file size
   rewind(fin);                      // reset the file pointer
   ```

   Use `fread` and `fwrite` routines to read and write the files. Look up on the Internet for usage examples if you are not familiar with them. A provided `datagen.c` program can be used to generate data files:

   ```
   linux> ./datagen 50 > data50   -- generate 50 integers and pipe to file 'data50'
   ```

3. MPI Scan and Scatter Routines

The files, `scan-mpi.c` and `scatter-mpi.c`, contain examples of using the MPI collective routines for scan and scatter. Read and understand these programs; then compile and run them.

The `scatter-mpi.c` program shows an example of the `MPI_Scatterv` routine, which scatters variable-sized sections of an array over processes:

```
// scatter variable-sized data sections to all processes
MPI_Scatterv(data, scnt, disp, MPI_INT, result, N, MPI_INT, 0, MPI_COMM_WORLD);
```

The second parameter `scnt` is a vector of *send_counts*, indicting the sections' size, and the third parameter `disp` is a vector of indices into the data array.

The Open MPI implementation on our Linux system has a small bug — it does not allow a *send_counts* element to be zero. Change the line `scnt[k] = k + 1;` (L44) to `scnt[k] = k;` and compile and run the program. You should see the program hangs.

You can fix this problem by changing the zero entry to something else, right before the `MPI_Scatterv` call. This should not affect the correctness of the program, since the original `scnt` value has already been scattered.