CSCI 3366 (Introduction to Parallel and Distributed Processing), Spring 2002

Homework 3

Assigned: February 19, 2002.

Due: February 28, 2002, at 5pm.

Credit: 40 points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

Contents

1 Overview **1**

2 Details **1**
- 2.1 The starter program 1
- 2.2 What you are to do, part 1: Coding 2
- 2.3 What you are to do, part 2: Running your program 2

3 Helpful hints **3**
- 3.1 The bucket sort algorithm 3
- 3.2 Some potentially useful STL functions 3
- 3.3 Cautionary comments and miscellaneous tips 3

4 What to turn in **4**

1 Overview

The textbook discusses an algorithm called "bucket sort" for using multiple processes to sort a list of numbers. For this assignment, your mission will be to implement this algorithm to sort an array of integers and compare its performance with that of a sequential (one-process) sort. Because reading numbers in and printing them out is tedious, we will generate the numbers to sort using a random number generator, and rather than printing them out we will compare the results to the results of sorting using a sequential sort. We will time only the part of the program that performs the bucket sort, allowing reasonable comparisons.

2 Details

2.1 The starter program

For this assignment, I have written an MPI main program that accepts as a command-line argument a number N representing the size of the array to sort. The program generates an array of N integers, sorts it, and compares the results to a correct sort. It also times the sorting step and

prints the results. The actual sorting is done in functions `masterProcess()` (running in process 0) and `slaveProcess()` (running in all other processes). As currently written, the `slaveProcess()` function does nothing, and all the work of sorting is done in `masterProcess()` — i.e., the program is currently a sequential sort.

You can obtain code for the starter program by downloading the following files:

- `bsort.cpp`[1] — code for the main program, plus function declarations for `masterProcess()` and `slaveProcess()`.
- `masterslave.cpp`[2] — definitions (code) for `masterProcess()` and `slaveProcess()`.
- `Makefile`[3] — a custom Makefile for the `bsort` program (not really essential, but convenient).

To compile the program, use the command

```
make bsort
```

Run the executable (named `bsort`) as you would run any other MPI program with a command-line argument; e.g., to run it with one process and sorting 1000 numbers, use this command:

```
mpirun -np 1 bsort 1000
```

The provided code also includes some examples of adding "debug print" statements that can be included/excluded at compile time (look up `#ifdef` in a C/C++ textbook to learn more about this). By default, these statements are included; to include them, compile as follows:

```
make DEBUG= bsort
```

You may find this feature useful for adding your own debug print statements; then you can easily turn them on (for debugging) and off (for collecting timing information) simply by varying the command you use to compile.

2.2 What you are to do, part 1: Coding

You are to replace the contents of `masterProcess()` and `slaveProcess()` with code to implement the bucket sort algorithm described in the textbook. You should *not* make changes in the main program; the main program as currently written tests the correctness of the sorting operation and so will serve as a demonstration that your code to perform the sort works properly. So, you will change only file `masterslave.cpp`, not `bsort.cpp`. Comments in file `masterslave.cpp` describe the parameters to the two functions you are to modify. Of course, you should feel free to add additional functions and/or `#include` statements. Be sure your program works for any number of processes (not just, for example, if the number of processes is a power of 2).

2.3 What you are to do, part 2: Running your program

Once you have your code producing correct results (as reported by the main program), run it for varying values of N and for varying numbers of processes, and compare execution times with execution times for the original program. Ideally execution time for your code using one process will be not much worse than execution time of the original program for the same value of N, and will decrease as you increase the number of processes. Also, your code should perform better relative to the original code for larger values of N. Experiment to see if this appears to be the case. Record at least half a dozen observations (timing results for sorting N numbers with P processes, compared to using the original program to sort N numbers).

[1] `http://www.cs.trinity.edu/~bmassing/CS3366_2002spring/Homeworks/HW03/Problems/bsort.cpp`
[2] `http://www.cs.trinity.edu/~bmassing/CS3366_2002spring/Homeworks/HW03/Problems/masterslave.cpp`
[3] `http://www.cs.trinity.edu/~bmassing/CS3366_2002spring/Homeworks/HW03/Problems/Makefile`

3 Helpful hints

3.1 The bucket sort algorithm

See chapter 4 of the textbook for a discussion of the algorithm. Two versions are presented, one in which each process gets a copy of all N numbers and one in which each process gets only N/P of them. You can choose to implement either version; my experiments suggest that the second one is more work to implement but performs somewhat better.

One thing to watch out for in doing such comparisons: You should be running both programs (your parallel one and the original one) on similar machines. That is, if you are logged into one of the Janus machines, but you are running the parallel program using the Dwarf machines, execution times may not be comparable.

3.2 Some potentially useful STL functions

Some of you may be familiar with the C++ Standard Template Library and its functions. The provided code makes use of two of them that you also may find useful:

- `sort()` sorts the elements of a container (the STL generic term for arrays, lists, vectors, etc.) into a desired order. To use it to sort array `nums` of size `N` in non-descending order, write

  ```
  sort(nums, nums+N)
  ```

- `copy()` copies elements from one container to another. To use it to copy `N` elements from array `source` to array `dest`, write

  ```
  sort(source, source+N, dest)
  ```

A good Web source of detailed information about this library is SGI's online Standard Template Library (STL) Programmer's Guide[4].

3.3 Cautionary comments and miscellaneous tips

- P (the number of processes) might not evenly divide N. Your code should be prepared to cope with this.

- Figuring out which bucket a particular number belongs in requires you to work with the minimum and maximum values allowed for the numbers. These are provided to function `masterProcess()` as parameters `minVal` and `maxVal`. Be aware that these numbers could be the smallest and largest possible integers, so it may not be possible to compute the number of possible values (`maxVal - minVal + 1`) using integer arithmetic. (Think about why not.) It is probably safest to do the needed calculations using `double`s and then round the result back to an `int` using the `ceil()` and/or `floor()` functions.

- In collecting timing observations, be sure all the machines you are using are about equally fast; otherwise comparisons between the original program and your parallel program will not be very meaningful. A way to do this is to be sure the machines you are using for the parallel program are the same "type" (Janusxx, Xenaxx, Dwarf$x$) as the machine you are using for the original program. Recall that you can specify what machines MPI is to use for running your parallel program by putting their names in a text file, say `MPIhosts`, and then telling

[4] `http://www.sgi.com/tech/stl/`

MPI to use that file; see Tips for Using MPI on the CS Linux Machines[5] for details of how to do this.

- Also note that in collecting timing information you will need two versions of the program `bsort` (the original one that does all sorting in one process, and your modified one). One way to avoid filename conflicts between the two programs is to put each in a separate directory, with each directory having a copy of files `bsort.cpp`, `masterslave.cpp`, `Makefile`, executable `bsort`, etc.

4 What to turn in

Submit your revised version of source file `masterslave.cpp`, plus a text file containing your timing observations (as described in the "Details" section above), by e-mail as described in the Guidelines for Programming Assignments[6], using a subject header of "cs3366 hw 3". You do not need to submit file `bsort.cpp` (you didn't change it anyway, right?). Please submit the timing observations as a plain text file.

[5] `http://www.cs.trinity.edu/~bmassing/CS3366_2002spring/Notes/tips-mpi/index.html`
[6] `http://www.cs.trinity.edu/~bmassing/CS3366_2002spring/Notes/pgmguidelines/index.html`