

## 1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework03, put each programming exercise into its own subdirectory of this directory, zip the entire Homework03 directory up into the file Homework03.zip, and then submit this zip file to Homework #3. Make sure all the code and document files are included in the zip file.

## 2 Programming Exercises

1. Take program number 3 from the ICE #2 exercise set and redo the analysis on your cluster. As a reminder, the exercise was the following.

Write an MPI program that will parallelize the calculation of a vector dot product (also called a scalar product). Recall that the dot product of two vectors is the sum of the products of the respective entries. For example, if you have two vectors from  $\mathbb{R}^4$ ,  $v = (x, y, z, w)$  and  $w = (a, b, c, d)$ , then  $v \cdot w = xa + yb + zc + wd$ . Have the user input the size of the vectors, have process 0 create two arrays of random numbers (doubles) between  $-1$  and  $1$ . Have the program,

- (a) Find the dot product of the two vectors serially, and time the process.
- (b) Find the dot product of the two vectors using a parallel implementation that uses only point-to-point communication, that is, just sends and receives. Also time the process.
- (c) Find the dot product of the two vectors using a parallel implementation that uses collective communication, that is, broadcasts and reductions. Also time the process.
- (d) The program should also check the results of the parallel implementations against the serial one. Of course, there will be some minor deviation due to round-off error and the order of the floating point arithmetic. So we will say that the two values are equal if the absolute value of the difference is less than  $0.000001$ .
- (e) Have the program calculate the speedup and efficiency of the run.

Now we will do an analysis of the program on the cluster you built. The instructions here are a little different than those in ICE #2 since you will be working with a “more” distributed system and multiple computers.

First, make several MPI host files (not system host files). One host file should use just a single worker node for computations and using the number of true cores of the machine. A second host file should use just a single worker node for computations and using the number of hardware threads of the machine. Then two more host files that use two machines in your cluster, one with just core usage and one with hardware

thread usage. A third set of two using three machines and a fourth set of two using all for worker nodes. You should also have a host file using the entire cluster for computation, one using cores and one using hardware threads. Most likely you have already created the last two.

- (a) Collect the data: Run this on various size arrays (big ones too) using different numbers of worker nodes. For example, run array size 1,000,000 for 1, 2, 3, and all 4 machines, then size 2,000,000 for 1, 2, 3, and all 4 machines and so on until you have data that is representable of the application speed. Do these timings for the two cases of just using machine cores and using hardware threads.
  - (b) Using the data from part 2a, make graphs of the speedup verses array size for each of the numbers of processors you were using. Then make graphs of the speedup verses the number of processors for each array size. For example, take all the processor numbers you ran with and array size of 1,000,000 and make a line (xy-scatter) graph with them. Do the same with array sizes of 2,000,000, 5,000,000 etc.
  - (c) Using your data, determine if your parallel application is strongly scalable, weakly scalable, follows Amdahl's Law, and follows Gustafson's Law. Justify all of your conclusions. If you need to run more test runs to make this determination do so and include those results in your report.
  - (d) Write up a short report on all of your findings from your analysis including all your data, graphs, conclusions, and justifications. Save your report in PDF format and include it in the zip file of all your code for this exercise.
2. You can never sort enough arrays ... Attached to this exercise set is a section from a Data Structures text on the Shell Sort. You probably saw the Shell Sort in COSC 220 or COSC 320, or possibly both. You may wish to review this to refresh your memory on how it works. The Shell Sort is easily parallelizable and is "almost" ideal for shared memory systems. Although we will be creating a distributed memory application for this, you may wish to think about how you could implement it using OpenMP.

You will be writing two implementations of the Shell Sort and testing them on your clusters. As you can gather from the description of the algorithm in the handout the three main questions in implementing this algorithm are

- (a) The sequence of increments
- (b) A simple sorting algorithm applied in all passes except the last
- (c) A simple sorting algorithm applied only in the last pass, for 1-sort

For the last two questions we will follow Shell's original algorithm and use the insertion sort. For the first question we will take two approaches.

- (a) For the first program you will use the  $h_{i+1} = 3h_i + 1$  method. This produces the sequence of increments of 1, 4, 13, 40, 121, 364, 1093, 3280, ... Now the handout

also mentions that we stop the sequence at  $h_t$  when  $h_{t+2} \geq n$ , where  $n$  is the size of the array. We will take a slightly different approach. Since the increment is also the number of subarrays that are individually sorted, and it makes the most sense to give each subarray to a processor, we will use the upper bound on the increment to be the number of processors we are using in the calculation. For example, if we are using all 56 cores of your cluster then we would use increments of 1, 4, 13, and 40. So the first pass would use 40 of our 56 cores and each core would get one of the 40 subarrays, then the second pass we would use 13 of our cores, third pass just 4 of them, and (of course) the last pass would use a single core. On the other hand, if we ran this on just two worker nodes (24 total cores) then we would use increments 1, 4, and 13. Your program should determine the maximum increment size for any communicator size as well as the entire list of increments.

- (b) The second program will be similar but the increments will start with the number of processes in the communicator, the next pass will use  $1/3$  of them (integer division), the next pass will use  $1/3$  of those, and so on until the last step where we use just one process. For example, say we are running all 56 cores. The first increment would be 56, the second would be 18, third would be 6, then 2, and finally 1. If we were running just the 48 on the worker nodes then the sequence would be 48, 16, 5, and then 1.

In addition,

- (a) Each program should also do the same sort with the same increments serially.
- (b) Time both the serial and parallel versions of each sort and computing speedup.
- (c) Each program should also display the speedup and efficiency of the parallel verses the serial run.
- (d) The programs should also test that each of the arrays is sorted and report if they are or are no in sorted order.
- (e) The programs should also compare the two arrays and determine if they are equal or not, and of course display the result to the console.
- (f) The arrays will just be arrays of integers. Let the range just be from 0 to the maximum int size (i.e. the default of the `rand()` function).
- (g) Let the user input the size of the array to be sorted and then just fill the array with the random numbers.
- (h) Also let the user select if they would like to print out the initial array and sorted array. This will also help you with testing your program.

Since these programs are very similar you can write them in a single program if you would like. Personally, I would do two separate ones so that the code is a bit more readable. If you decide to do this in a single program, put barriers between the sorting methods to guarantee that all processes are starting and stopping where they should before proceeding on to the next timing test. For the analysis, use each program to,

- (a) Collect the data: Run this on various size arrays (big ones too) using different numbers of worker nodes. That is, run each array size for 1, 2, 3, and all 4 machines. Get enough data to give a representable indication of the application speed. Do these timings for the two cases of just using machine cores and using hardware threads.
- (b) Using the data from part 2a, make graphs of the speedup verses array size for each of the numbers of processors you were using. Then make graphs of the speedup verses the number of processors for each array size.
- (c) Using your data, determine if your parallel application is strongly scalable, weakly scalable, follows Amdahl's Law, and follows Gustafson's Law. Justify all of your conclusions. If you need to run more test runs to make this determination do so and include those results in your report.
- (d) Write up a short report on all of your findings from your analysis including all your data, graphs, conclusions, and justifications. Save your report in PDF format and include it in the zip file of all your code for this exercise.