

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.

of 188 and 376, respectively. This can only serve to encourage the search for an algorithm embodying the performance of the function $n \lg n$.

9.3 EFFICIENT SORTING ALGORITHMS

9.3.1 Shell Sort

The $O(n^2)$ limit for a sorting method is much too large and must be broken to improve efficiency and decrease run time. How can this be done? The problem is that the time required for ordering an array by the three sorting algorithms usually grows faster than the size of the array. In fact, it is customarily a quadratic function of that size. It may turn out to be more efficient to sort parts of the original array first and then, if they are at least partially ordered, to sort the entire array. If the subarrays are already sorted, we are that much closer to the best case of an ordered array than initially. A general outline of such a procedure is as follows:

```

divide data into h subarrays;
for i = 1 to h
    sort subarray datai;
sort array data;

```

If h is too small, then the subarrays data_i of array data could be too large, and sorting algorithms might prove inefficient as well. On the other hand, if h is too large, then too many small subarrays are created, and although they are sorted, it does not substantially change the overall order of data . Lastly, if only one such partition of data is done, the gain on the execution time may be rather modest. To solve that problem, several different subdivisions are used, and for every subdivision, the same procedure is applied separately, as in:

```

determine numbers  $h_t \dots h_1$  of ways of dividing array data into subarrays;
for (h=ht; t > 1; t--, h=ht)
    divide data into h subarrays;
    for i = 1 to h
        sort subarray datai;
sort array data;

```

This idea is the basis of the *diminishing increment sort*, also known as *Shell sort* and named after Donald L. Shell who designed this technique. Note that this pseudo-code does not identify a specific sorting method for ordering the subarrays; it can be any simple method. Usually, however, Shell sort uses insertion sort.

The heart of Shell sort is an ingenious division of the array data into several subarrays. The trick is that elements spaced farther apart are compared first, then the elements closer to each other are compared, and so on, until adjacent elements are compared on the last pass. The original array is logically subdivided into subarrays by picking every h_t th element as part of one subarray. Therefore, there are h_t subarrays, and for every $h = 1, \dots, h_t$,

$$\text{data}_{h_t h}[i] = \text{data}[h_t \cdot i + (h-1)]$$

For example, if $h_t = 3$, the array `data` is subdivided into three subarrays `data1`, `data2`, and `data3` so that

```
data31[0] = data[0], data31[1] = data[3], ..., data31[i] = data[3*i], ...
data32[0] = data[1], data32[1] = data[4], ..., data32[i] = data[3*i+1], ...
data33[0] = data[2], data33[1] = data[5], ..., data33[i] = data[3*i+2], ...
```

and these subarrays are sorted separately. After that, new subarrays are created with an $h_{t-1} < h_t$ and insertion sort is applied to them. The process is repeated until no subdivisions can be made. If $h_t = 5$, the process of extracting subarrays and sorting them is called a 5-sort.

Figure 9.7 shows the elements of the array `data` that are five positions apart and are logically inserted into a separate array—“logically” because physically they still occupy the same positions in `data`. For each value of increment h_t , there are h_t subarrays, and each of them is sorted separately. As the value of the increment decreases, the number of subarrays decreases accordingly, and their sizes grow. Much of `data`’s disorder has been removed in the earlier iterations, so on the last pass, the array is much closer to its final form than before all the intermediate h -sorts.

FIGURE 9.7 The array [10 8 6 20 4 3 22 1 0 15 16] sorted by Shell sort.

data before 5-sort	10	8	6	20	4	3	22	1	0	15	16
Five subarrays before sorting	10	—	—	—	—	3	—	—	—	—	16
		8	—	—	—	—	22	—	—	—	
			6	—	—	—	—	1	—	—	
				20	—	—	—	—	0	—	
					4	—	—	—	—	15	
Five subarrays after sorting	3	—	—	—	—	10	—	—	—	—	16
		8	—	—	—	—	22	—	—	—	
			1	—	—	—	—	6	—	—	
				0	—	—	—	—	20	—	
					4	—	—	—	—	15	
data after 5-sort and before 3-sort	3	8	1	0	4	10	22	6	20	15	16
Three subarrays before sorting	3	—	—	0	—	—	22	—	—	15	
		8	—	—	4	—	—	6	—	—	16
			1	—	—	10	—	—	20	—	
Three subarrays after sorting	0	—	—	3	—	—	15	—	—	22	
		4	—	—	6	—	—	8	—	—	16
			1	—	—	10	—	—	20	—	
data after 3-sort and before 1-sort	0	4	1	3	6	10	15	8	20	22	16
data after 1-sort	0	1	3	4	6	8	10	15	16	20	22

One problem still has to be addressed, namely, choosing the optimal value of the increment. In the example in Figure 9.7, the value of 5 is chosen to begin with, then 3, and 1 is used for the final sort. But why these values? Unfortunately, no convincing answer can be given. In fact, any decreasing sequence of increments can be used as long as the last one, h_1 , is equal to 1. Donald Knuth has shown that even if there are only two increments, $(\frac{16n}{\pi})^{\frac{1}{3}}$ and 1, Shell sort is more efficient than insertion sort because it takes $O(n^{\frac{5}{3}})$ time instead of $O(n^2)$. But the efficiency of Shell sort can be improved by using a larger number of increments. It is imprudent, however, to use sequences of increments such as 1, 2, 4, 8, . . . or 1, 3, 6, 9, . . . because the mixing effect of data is lost.

For example, when using 4-sort and 2-sort, a subarray, $\text{data}_{2,i}$, for $i = 1, 2$, consists of elements of two arrays, $\text{data}_{4,i}$ and $\text{data}_{4,j}$, where $j = i + 2$, and only those. It is much better if elements of $\text{data}_{4,i}$ do not meet together again in the same array because a faster reduction in the number of exchange inversions is achieved if they are sent to different arrays when performing the 2-sort. Using only powers of 2 for the increments, as in Shell's original algorithm, the items in the even and odd positions of the array do not interact until the last pass, when the increment equals 1. This is where the mixing effect (or lack thereof) comes into play. But there is no formal proof indicating which sequence of increments is optimal. Extensive empirical studies along with some theoretical considerations suggest that it is a good idea to choose increments satisfying the conditions

$$h_1 = 1$$

$$h_{i+1} = 3h_i + 1$$

and stop with h_t for which $h_{t+2} \geq n$. For $n = 10,000$, this gives the sequence

$$1, 4, 13, 40, 121, 364, 1093, 3280$$

Experimental data have been approximated by the exponential function, the estimate, $1.21n^{\frac{5}{3}}$, and the logarithmic function $.39n \ln^2 n - 2.33n \ln n = O(n \ln^2 n)$. The first form fits the results of the tests better. $1.21n^{1.25} = O(n^{1.25})$ is much better than $O(n^2)$ for insertion sort, but it is still much greater than the expected $O(n \lg n)$ performance.

Figure 9.8 contains a function to sort the array `data` using Shell sort. Note that before sorting starts, increments are computed and stored in the array `increments`.

The core of Shell sort is to divide an array into subarrays by taking elements h positions apart. Three features of this algorithm vary from one implementation to another:

1. The sequence of increments
2. A simple sorting algorithm applied in all passes except the last
3. A simple sorting algorithm applied only in the last pass, for 1-sort

In our implementation, as in Shell's, insertion sort is applied in all h -sorts, but other sorting algorithms can be used. For example, Incerpi and Sedgewick use two

FIGURE 9.8 Implementation of Shell sort.

```

template<class T>
void Shellsort(T data[], int n) {
    register int i, j, hCnt, h;
    int increments[20], k;
    // create an appropriate number of increments h
    for (h = 1, i = 0; h < n; i++) {
        increments[i] = h;
        h = 3*h + 1;
    }
    // loop on the number of different increments h
    for (i--; i >= 0; i--) {
        h = increments[i];
        // loop on the number of subarrays h-sorted in ith pass
        for (hCnt = h; hCnt < 2*h; hCnt++) {
            // insertion sort for subarray containing every hth element of
            for (j = hCnt; j < n; ) { // array data
                T tmp = data[j];
                k = j;
                while (k-h >= 0 && tmp < data[k-h]) {
                    data[k] = data[k-h];
                    k -= h;
                }
                data[k] = tmp;
                j += h;
            }
        }
    }
}

```

iterations of cocktail shaker sort and a version of bubble sort in each h -sort and finish with insertion sort, obtaining what they call a *shakersort*. All these versions perform better than simple sorting methods, although there are some differences in performance among versions. Analytical results concerning the complexity of these sorts are not available. All results regarding complexity are of an empirical nature.

9.3.2 Heap Sort

Selection sort makes $O(n^2)$ comparisons and is very inefficient, especially for large n . But it performs relatively few moves. If the comparison part of the algorithm is improved, the end results can be promising.