1. **Problem Statement**

Generate 128,000,000 32-bit numbers on each of 32 nodes and sort them using the Parallel Sorting with Regular Sampling (PSRS) algorithm. Sorting needs to be verified by the parallel definition of sorted. The program is timed on based on the sorting tasks performed. Design and implement a program with as optimal performance as possible using MPI between processes and OpenMP for multi-threading within a process. Document performance and include a Karp-Flatt metric.

1. **Approach**

The PSRS algorithm is great choice for sorting very large problem domains utilizing quicksort, one of the fastest sorting algorithms in the average case. PSRS also has the advantage of maintaining a balanced workload between processors, excelling at this over both parallel quicksort and hyperquicksort.

The best approach to this problem is to create a base Makefile for running the program, then implement all the requirements for a fully functional program in an incremental fashion. Through test-driven design, I can build and test incrementally as shown below in Figure 1.

**Test-driven Design Steps**

1. Makefile
2. C source with MPI setup
3. Command-line interface
4. Debug mode
5. Local sample collection populated with RNG
6. Quicksort subroutine
7. Local sample sort and local regular samples
8. MPI\_Gather
9. Collective sample sort and selected split values
10. MPI\_Bcast
11. Create send count array and allocate space for receive count array
12. MPI\_Alltoall
13. Create send and receive displacement arrays
14. MPI\_Alltoallv
15. Final quicksort
16. Timer (Profiling) mode
17. Profile the program
18. OpenMP optimizations
19. Final testing and cleanup

Figure . Steps to completing the program.

Following this implementation list, features are to be developed and tested along the way. Debug mode, which is triggered at the command-line, is to be set up fairly early in the process to allow debug testing throughout the software development lifecycle. The quicksort subroutine, itself, can be decomposed into three primary steps:

1. Quicksort (top level with recursive call)
2. Subsort (implementation level)
3. Swap (supporting swap subroutine)

After step 15, the program should be fully functional. At this point, the profiling mode with timers throughout the program, should be implemented. This allows for the program to be tested in profiling mode for performance testing and determining where the program would most benefit from OpenMP pragmas within a processor. Once this is all done, the program can be run with different number of processors to analyze the Karp Flatt metric for scalability analysis.

1. **Solution**

Following the implementation list in Figure 1 and testing along the way, I was able to complete a fully working program that met all requirements. The most problematic steps involved troubleshooting the quicksort subroutines which were used throughout the program. To build and run the program, the following make commands can be called using the Makefile:

**Makefile Commands**

**make**  make executable

**make test** test all test cases

**make debug** run in debug mode to show correct output

**make time** run program in profiling mode to show times

Figure . Makefile commands to test the program

Call **make** to simply build the executable. Calling **make test** tests a series of cases using the same problem size for process count 1-10. Calling **make debug** will execute a simple version of the program which uses sample values 0-29 for easier reading and verification. This prints out all the debug statements showing the logic every step of the way and illustrating correct functionality. Calling **make time** will run the program in profiling mode to show the execution time in each of the sections of the program. This was a valuable feature to determine where to apply OpenMP.

The quicksort subroutine works by accepting an initial array with the low and high index provided. The subroutine then sorts everything between these two indices by sending these operands to a subsort routine. Subsort uses the high index value as a pivot value. By incrementing through the entire array, subsort checks each element and compares it with the pivot value. If equal to or less than the pivot value, subsort swaps it to the lower end of the array using a subsort routine, keeping track of where the top of the lower end is. When finished, the pivot value is placed at the top of the lower end in its final resting place and that index is reported back to quicksort. Quicksort then takes everything to the left and right of that index and recursively calls quicksort.

After completing the MPI implementation of the PSRS algorithm, the final step was to determine where to apply OpenMP to further leverage shared-memory parallelization within a processor. In Figure 3, the program was tested running with 4 processors with 10,000 samples each. This was run prior to implementing OpenMP.

|  |  |
| --- | --- |
| **Profiling MPI Program for OpenMP Optimization** | |
| ***Run: N = 4, Samples = 10000/each*** |  |
| Quicksort initial collection | **0.02288** |
| Get regular samples | 0.00002 |
| Gather all reg samples | 0.00004 |
| Quicksort and select split values |  |
| from all reg samples | 0.00001 |
| Broadcast split values | 0.00004 |
| Calculate send counts | 0.00048 |
| Send all-to-all | 0.00005 |
| Calc send/recv displ | 0.00002 |
| Send all-to-allv | 0.00083 |
| Final quicksort | **8.45632** |
| Final time | 8.48072 |

Figure . Profiling the MPI program prior to OpenMP implementation.

As shown in Figure 3, the serial execution that would benefit the most from OpenMP parallelization would be the quicksort subroutines. This makes sense, since this part of the code is the most intensive work happens with swaps and recursive calls to quicksort. By applying OpenMP to this part of the code, some improvement was observed (Figure 4).

|  |  |
| --- | --- |
| **Profiling MPI + OpenMP added to Quicksort Routine** | |
| ***Run: N = 4, Samples = 100000/each*** |  |
| Quicksort initial collection | 0.02902 |
| Get regular samples | 0.00003 |
| Gather all reg samples | 0.00003 |
| Quicksort and select split values |  |
| from all reg samples | 0.00000 |
| Broadcast split values | 0.00005 |
| Calculate send counts | 0.00034 |
| Send all-to-all | 0.00003 |
| Calc send/recv displ | 0.00002 |
| Send all-to-allv | 0.00054 |
| Final quicksort | 8.01888 |
| Final time | 8.04895 |
| **Speedup** | **1.053643** |

Figure . Profiling the MPI program with OpenMP added to Quicksort Routine.

The improvement varied depending on the test case, but a noticeable improvement was made. Further testing could perhaps yield further improvements by minimizing the forking overhead and assigning recursive tasks to available threads. This could be done by using the forking omp parallel directive outside of the quicksort subroutine and assigning the two recursive quicksort calls to available threads as tasks.

To verify the correct execution of the program, debug statements were used throughout the program. As mentioned in Figure 2, **make debug** executes the program in debug mode producing the following print statements. Figure 5 shows the execution of the program with 4 processors running 8 samples each. As mentioned, because it is in debug mode, samples are simplified to values ranging 0-29 for easy reading. Figure 5 shows each process id with the initial collection of samples. The second printout in Figure 5 also shows the collection after the quicksort routine was executed.

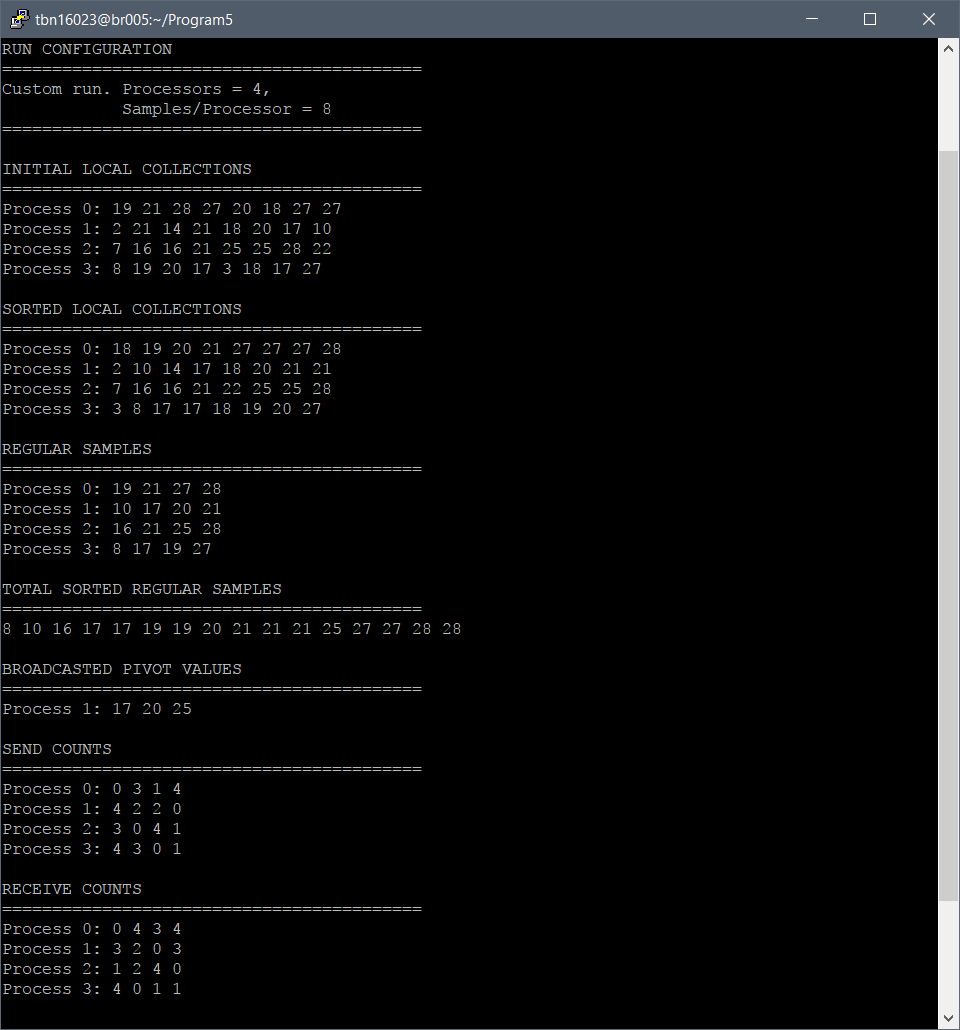


Figure . Debug mode printout showing initial and sorted local collections.

In the following print statements, Figure 6 shows the regular samples selected from each processor. Samples were collected and gathered using MPI\_Gather onto process 0. Then, using quicksort, they were sorted as illustrated in the second printout in Figure 6. Finally, several evenly spaced samples were selected from this collective sample equal to the number of processors – 1. This was broadcasted back to each process using MPI\_Bcast and one of the processors, process 1, illustrates the broadcasted list in the final printout in Figure 6.

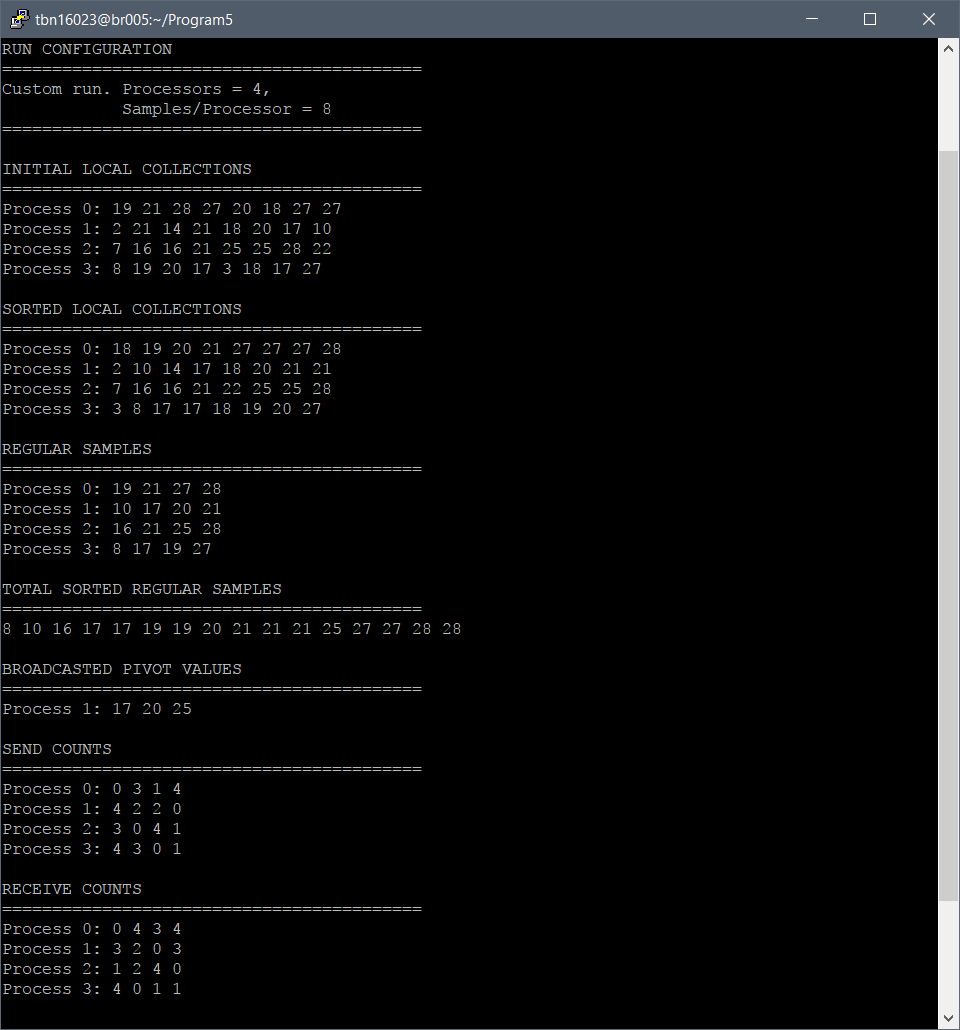


Figure . Debug mode printout showing regular sampling of local collections.

Next, in Figure 7, the send count array was constructed using the regular samples received as split values to divide the local collections. Using this array, an all-to-all exchange was made using MPI\_Alltoall, so that all processors got how many values they will receive from each other in a receive counts array. Using this, space can be allocated for the final buffer and the send and receive displacement array could be made. The send and receive count arrays can be found below in Figure 7.

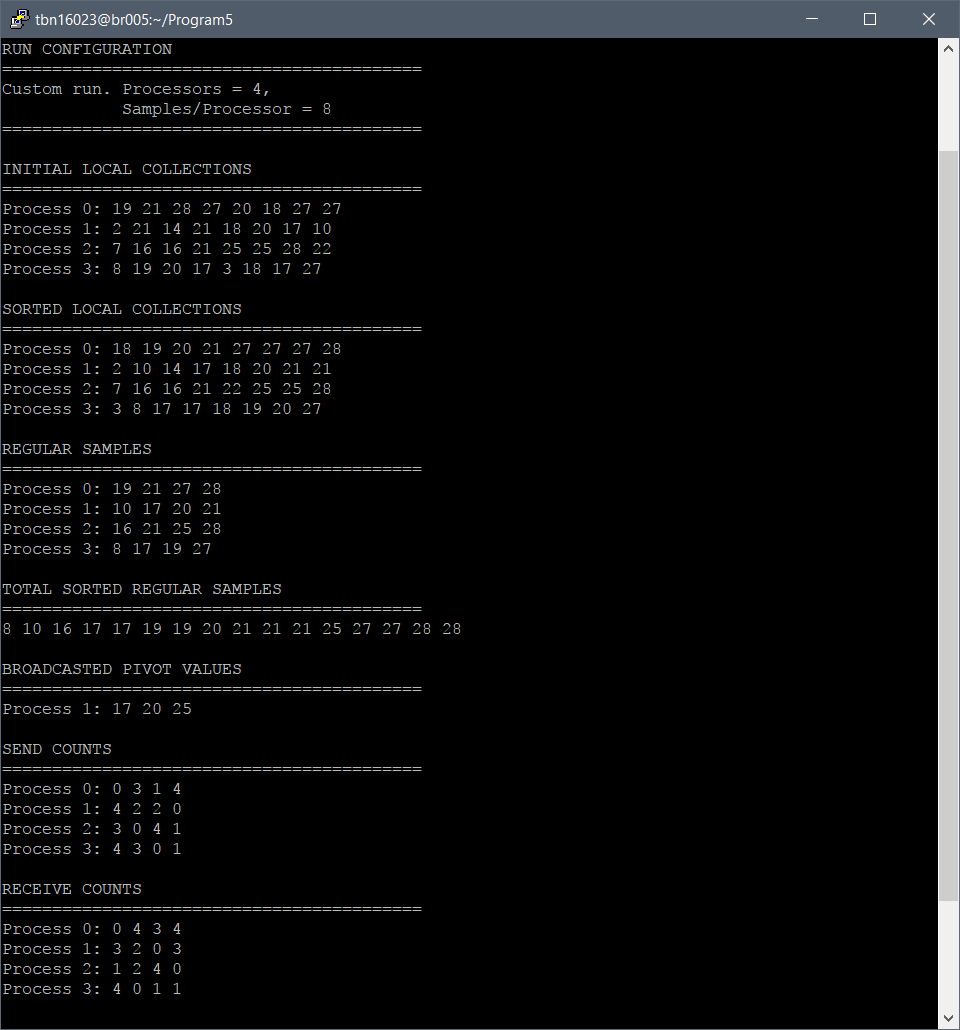


Figure . Debug mode printout showing the send/receive count arrays.

Finally, using all the arrays created before and the final buffer created, all processes do a final, variable all-to-all exchange using MPI\_Alltoallv. With each process having their final values, they all simply need to do a final quicksort satisfying the parallel definition of sorted. The final sorted arrays can be found in Figure 8.

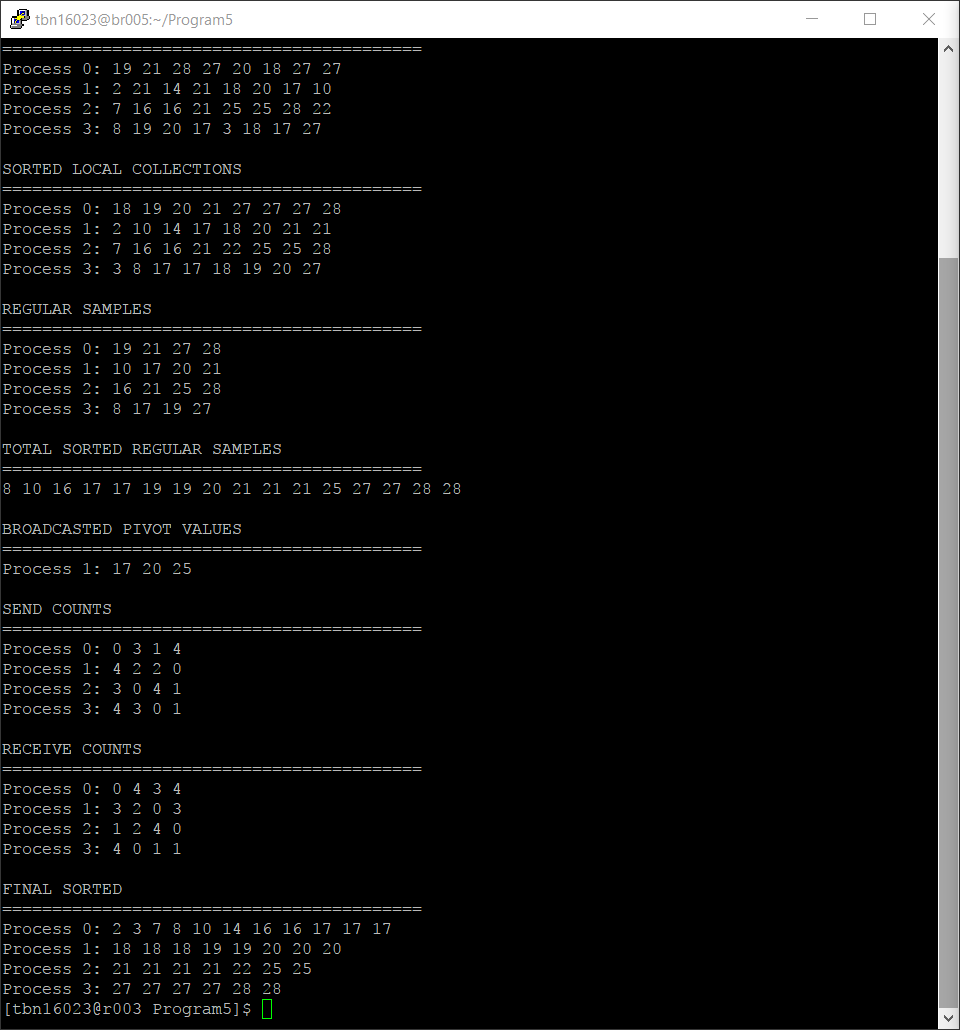


Figure . Debug mode printout showing the final sorted array in each processor.

With the program working and MPI + OpenMP parallelization implementation completed, the program can finally be tested to analyze performance and scalability. As mentioned, **make test** can be used to run the ten test cases used. The results in Figure 9 summarize these test cases.

|  |  |
| --- | --- |
| **Process Count** | **Time (sec)** |
| 1 | 28.28 |
| 2 | 7.11 |
| 3 | 3.18 |
| 4 | 1.79 |
| 5 | 1.20 |
| 6 | 0.82 |
| 7 | 0.63 |
| 8 | 0.48 |
| 9 | 0.38 |
| 10 | 0.33 |

Figure . Test results for PSRS program running 100,000 samples across 1-10 processes.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Processors** | **p** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **Time** | **t** | 28.28 | 7.11 | 3.18 | 1.79 | 1.20 | 0.82 | 0.63 | 0.48 | 0.38 | 0.33 |
| **Speedup** | **y** | 1.00 | 3.98 | 8.90 | 15.78 | 23.55 | 34.28 | 45.17 | 58.76 | 74.28 | 85.82 |
| **Experimental** | **e** | - | -0.50 | -0.33 | -0.25 | -0.20 | -0.16 | -0.14 | -0.12 | -0.11 | -0.10 |
| **Serial Fraction** |  |  |  |  |  |  |  |  |  |  |  |

Running calculations on these experimental results, the speedup and Karp Flatt metric can be calculated as shown in Figure 10.

Figure . Speedup and experimental serial fraction (Karp Flatt metric) calculations.

As illustrated in Figure 11, the speedup continues to grow as the number of processors increase with some indication of “elbowing out” at greater than 10 processors. The execution time is shown to drop dramatically between 2-4 processors which can be seen in Figure 12.

Figure . Illustration of the speedup observed from 1-10 processors operating on 100,000 samples.

Figure . Illustration of the execution time observed from 1-10 processors operating on 100,000 samples.

All of these results can be found in the attached Excel spreadsheets **Results.xlsx**. Analyzing the experimental determined serial fraction in Figure 10, because the serial fraction is steadily increasing, communication overhead is the primary reason preventing further increase in performance. As such, to best improve this program further, the overall structure of the program and how it handles MPI communications would be the best place to dedicate time to for further optimization.