Shared-Memory Programming: Parallel Sorting by Regular Sampling

Student: Qasim Khawaja (1571051)

Introduction

This assignment aimed to apply the PSRS algorithm in an SPMD-style parallel program, thus gaining experience with parallel programming. This report covers the lessons learned trying to ensure correctness when writing a parallel program, assessing performance using well-designed tests, and lessons for crafting performant parallel programs. Overall, this report aims to underscore these three key points. Appropriate granularity, optimizing for the common case, and reducing data redundancy are crucial for creating more performant parallel algorithms.

Implementation

The following experiments were run on a 32 CPU and 32 GB Memory Linux system [Fig. 1]. The implementation is in C++ using an SPMD configuration, which uses pthreads and barrier programming for coordinating the work among the threads. The program is compiled using the maximum optimization setting using the -O3 flag. Global memory allocation was also used for the shared resources between the processors for the following objects. [Fig. 2]

```
struct StartEnd {
    long long start;
    long long end;
};

pthread_t* threads;
long* A;
long* samples;
long* pivots;
struct StartEnd* partitionStartEnd;
int p;
long long n;
int s;
```

Figure 2. Globally allocated data

Timing

The timing measurements use Chronos steady_clock and are measured in nanoseconds. Timing for each phase is obtained by setting a macro. [Fig. 3]

Generating Keys

The keys generated were of type long, and various techniques were used to generate the keys. By default, the program uses srandom to generate a uniform distribution. Another method used was to generate a normal distribution centred around 500 with a mu of 100 and a uniform distribution with the range [400, 600]. The purpose was to analyze performance differences between normal and uniform; the results of this experiment are explained later in this paper.

Algorithmic Details

This implementation uses binary search to find the partition boundaries; these boundaries are stored in global memory with a start and end index mapping to the global array that's being sorted. The phase 4 merge of these sorted partitions is accomplished using a k-way merge algorithm that uses a heap data structure. This implementation uses an SPMD-style approach to parallel programming, where each thread uses the same PSRS program, and synchronization is achieved using pthread barriers. One advantage of this approach is that it ensures data locality, since each thread accesses the same global data instead of communicating over a MPI, and because this program naturally fits into this structure where the phases can be separated using barriers.

Verifying Correctness

One of the most critical challenges with writing parallel programs is ensuring that the program produces the correct output. The output was carefully validated for this program through various experiments and tests. These tests for correctness are examined in the following sub-sections dealing with invariants, examining small examples, and expanding the validation to larger examples.

Invariants

Assertions are a programming technique that allows the programmer to use invariants to design the algorithm. In this implementation, a few invariants were used to ensure that the program behaved as expected with the given parameters. Some examples of these invariants included validating that number of processors is less than the number of elements cubed. As well as checking if each thread outputs a sorted list. While developing, assertions were also used to validate the algorithm's correctness throughout the different phases, for example, to ensure that each thread's start and end index was valid or to check if the partition boundaries were correct. These were inevitably not included in the final code since other tests for correctness were used to validate the final output.

Examining Small Examples

Another technique that proved helpful for checking that the program behaved as expected was to run the program on small worked examples, for example, the example in the PSRS paper [Li et al., 1993], printing each thread sorted list and comparing it to the example. It was essential to design a general testing method to ensure that the program behaves correctly for novel examples. This was achieved using C++ macros to print the unsorted list initially, then print each thread's sorted list. By combing their output and sorting it, it was possible to test if the program's final output contains all the keys and is sorted correctly.

Expanding to Larger Examples

For large examples where printing to the terminal is impossible, another test was designed to store each thread's maximum and minimum key and the counts. After verifying that each thread is in sorted order, the threads each push an object containing their min, max, and count to the main process using mutual exclusion to push to the global vector. Then the main thread compares the local sorted arrays to ensure that they are disjoint and the total number of keys adds up to n, where n is the number of elements in the target list.

Experimental Setup

The experimental study was conducted in python using a Jupyter notebook. The purpose of these experiments was to compare the performance of this program across different configurations and scenarios. A total of seven runs were completed for each data point in the tests, and the average of the last five runs was used in the analysis and graphs. This section describes the types of experiments run, and later sections will examine the results of these experiments.

Speedups and Relative times
The first set of experiments focused on speedups and relative times for different sizes and variating threads. For this experiment, the program was run on 10M, 20M, 40M, 80M, 160M, 320M, and 640M keys generated using the default srandom pseudo-random number generator. The program was run seven times for each thread from 1 to 16, and the average of the last five runs was used for the absolute time. This experiment aimed to understand how the program performs on different problem sizes with a different number of threads used.

Phase-by-Phase time breakdown
These experiments focused on the program's time used in the four phases. The program was run on 1M, 2M, 4M, 8M, 16M, 100M, 200M, and 400M keys using the default srandom pseudo-random number generator. This purpose was to understand what phase used the most time and any bottlenecks in the system.

Regular vs random sampling effect on load balancing
These experiments focused on the performance differences between regular sampling vs random sampling. This setup collected similar speedup and time data as the first set of experiments; however, limiting it to 8 threads. This also used both random and regular sampling using C++ macros. These experiments aimed to understand how regular sampling performs relative to random sampling.

Uniform vs Normal Distribution
These experiments focused on the time impact of using a normal vs uniform distribution on performance**.** Collected data for 200M elements using the uniform distribution in the range 400-600 and the normal distribution with a mu of 500 and a deviation of 100. To understand how the program performs with differently distributed data and whether the regular sampling performs the same for both.

Performance
The speedup graphs obtained from the experiments [Fig. 4 and 5] show an upward trend with adding additional threads; however, it can be observed that for a smaller number of keys, there are diminishing returns. This is directly related to the lesson about granularity since, for the smaller number of keys, the portion of work that an individual processor deals with is not significant enough. It can also be observed that the speedup was not linear; this can be explained by the phase-by-phase timing breakdown, which shows that phase 4 carries some overhead for this size of keys. It may be likely that with a larger number of keys, the effects of phase 4 on the speedup and timing will be diminished since its runtime is asymptomatically less than the runtime of quicksort in phase 1.

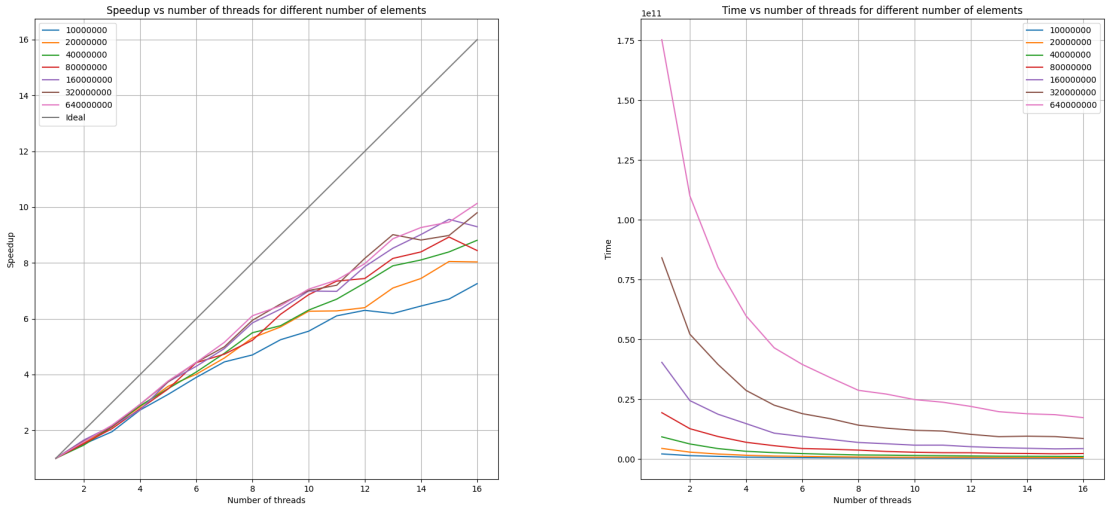

Figure 4, 5. Show the speedup and absolute time graph from running psrs on 10M to 640M elements with 1-16 threads.

Bottlenecks

The results from the phase-by-phase timing experiments [Fig 5,7] were used to understand what phases dominated the program's runtime and how they were changed when more threads were used. The first phase that used the most time was phase 1, which sorts the local partition for each array. This was roughly equivalent to the total time divided by the number of processors. The second phase that used the most time was phase 4, which combines the sorted partitions. The timing for Phase 2, although negligible in this setup since it is only run on one processor, does become a bottleneck as more threads are added and more samples need to be sorted. It is also interesting to note that the time used by phase 4 relative to the total time decreases as the total number of elements increases. This is directly related to the effects of granularity since the problem size needs to be big enough to make the parallel implementation more performant.

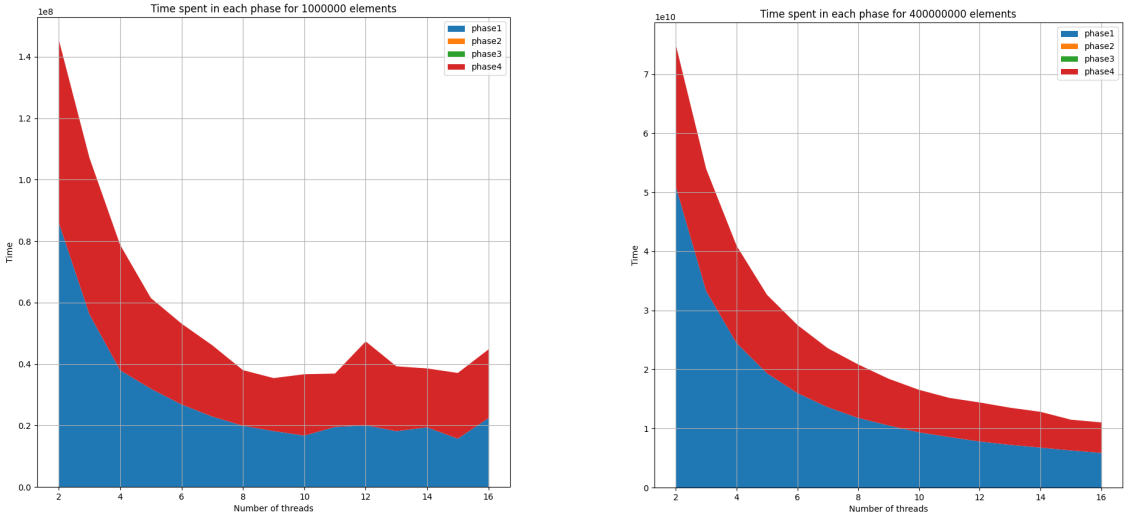

Figure 6,7. Shows the time spend in each phase for 10M elements and 400M elements respectively.

Regular Sampling impact on Load Balancing

The charts show a difference between the speedup and time for regular and random sampling [Fig 8,9,10,11.] It can be observed that regular sampling uses less time as the number of threads increases and has a more positive speedup trend. This result may be because regular sampling leads to more representative pivots that lead to better partitioning and load balancing. Another attempted experiment compared the algorithm's performance using a normal and uniform distribution. This experiment was inconclusive because there was not enough difference between the uniform and non-uniform distribution. Testing this out with random sampling would be an interesting follow-up experiment to understand the effects of regular sampling on maintaining an equal distribution of work among the threads.

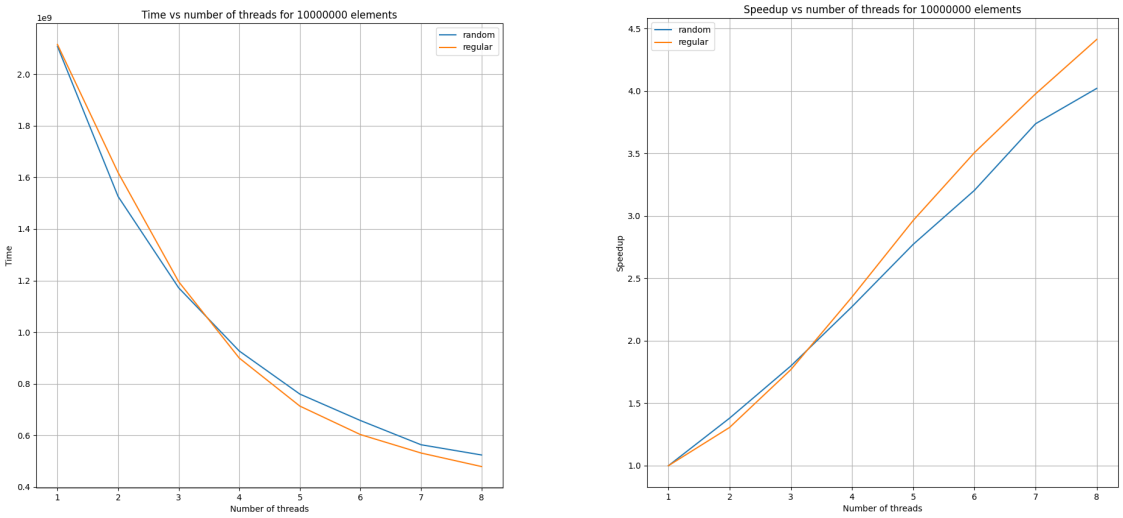

Figure 8, 9. Speedup and time graph for 10M elements using both random and regular sampling.

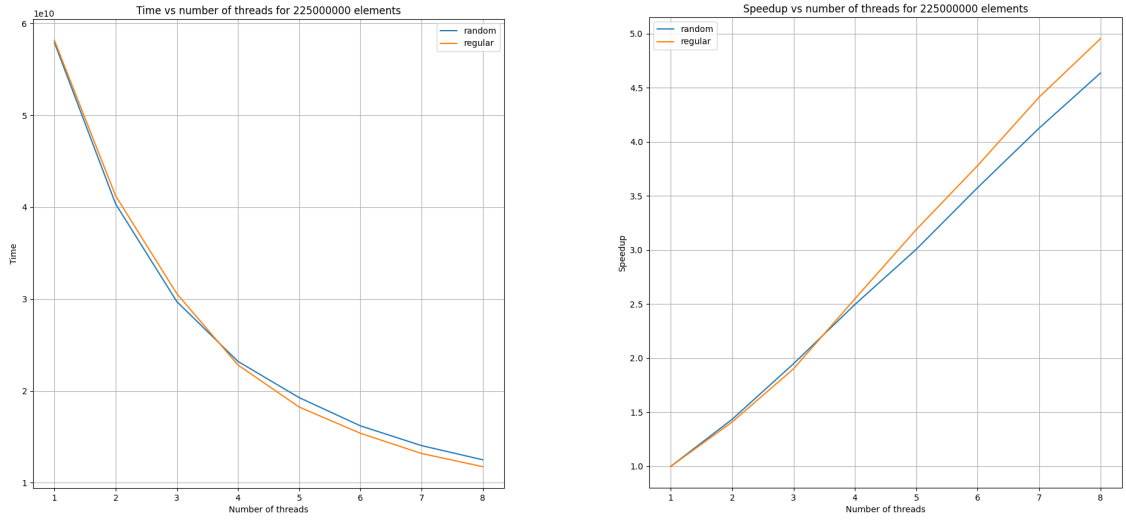

Figure 10, 11. Speedup and time graph for 225M elements using both random and regular sampling.

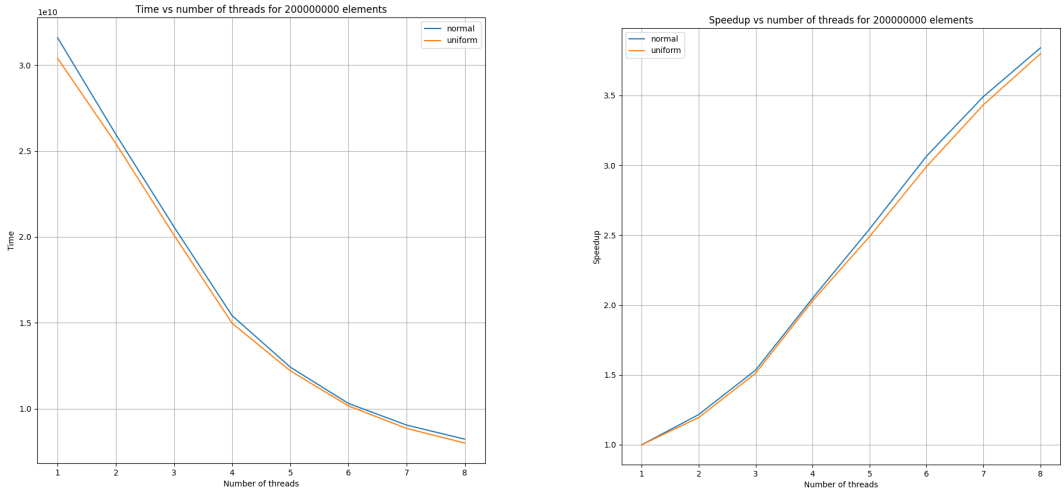

Figure 11, 13. Speedup and time graph for 225M elements using both uniform and normal distribution and regular sampling.

Optimizing for the common case

Earlier in the process of implementing this program, there was some effort spent trying to optimize phases 2 and 3 to improve the speedup; however, when running the performance benchmarks, it was clear that there was no advantage to doing this. This is a perfect example of why optimizing for the common case is important; when phase 4 was changed to use a k-way merge using a heap instead of comparing each partition one by one, there was a considerable improvement in the speedup. This is Ahmdaals law in effect, where the infrequently accessed portion of the code benefitted little from performance speedup.

Data Redundancy

Since this program was in a shared memory configuration, there was no need to copy data from the different threads, which is why phase 3 and phase 2 had minimal impact on the performance.

Conclusion

This report highlighted the importance of correctness, performance and granularity when parallel programming. All of these concepts are essential for creating efficient and correct algorithms. Furthermore, the report also aimed to provide intuition about how regular sampling can lead to better load balancing and performance in a parallel program. Overall, these lessons are crucial for parallel programming and shared-memory programming.

Sources

Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P. S., & Shi, H. (1993). On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, *19*, 1079–1103.

House, D. H. (n.d.). *Donald H. House, Homepage*. Retrieved October 10, 2022, from https://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf

Figures

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          32
On-line CPU(s) list:             0-31
Thread(s) per core:              2
Core(s) per socket:              16
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           79
Model name:                      Intel(R) Xeon(R) CPU @ 2.20GHz
Stepping:                        0
CPU MHz:                         2199.998
BogoMIPS:                        4399.99
Hypervisor vendor:               KVM
Virtualization type:             full
L1d cache:                       512 KiB
L1i cache:                       512 KiB
L2 cache:                        4 MiB
L3 cache:                        55 MiB
NUMA node0 CPU(s):               0-31
```

Figure 1. Computer hardware configuration used in this report

```
#if TIMING
std::chrono::_V2::steady_clock::time_point beginPhaseTime;
std::chrono::_V2::steady_clock::time_point endPhaseTime;
#define START_TIMER beginPhaseTime = std::chrono::steady_clock::now();
```

```
#define END_TIMER endPhaseTime = std::chrono::steady_clock::now();
#define PRINT_TIME(threadId, phase) \
    std::cout << "Thread: " + std::to_string(threadId) + " Phase: " +
std::to_string(phase) + " time: " +
std::to_string(std::chrono::duration_cast<std::chrono::nanoseconds>(endPhas
eTime - beginPhaseTime).count()) + "\n";
#else
#define START_TIMER
#define END_TIMER
#define PRINT_TIME(threadId, phase)
#endif
```

Figure 3. Macros used for timing