PSRS Report

Junwen Shen
1770012
junwen5@ualberta.ca

October 8, 2022

Abstract

Parallel sorting by regular sampling[6] is a parallel sorting design that uses regular sampling to balance the workload between threads. In this assignment, I implement the algorithm using *C++* and *pthread* and do a benchmark and analysis on the implementation.

1 PSRS Implementations

Usually, engineers separate their *C++* programs into header files and implementation files according to the design pattern of *C++*. Thus, I also follow the same pattern to design the implementation of PSRS, where only public interfaces are exported in header files, and other functions are placed in *.cpp* files, decorated by the static key word. The program contains two *namespace*s (*util* and *psrs*) and the entry point *main.cpp*, which is for test purposes only. In the following sub-section, I introduce each *namespace* in details.

1.1 utils

This *namespace* is used for providing a utility class, *Timer*[1], and two helper functions, namely, *print_vector*[9] and *format*[5]. The *Timer* class is a template class that can let users measure duration by calling *start* and *stop* methods on its instance. Moreover, it is a *std::chrono::steady_clock* internally, which is most suitable for measuring intervals[3]. *print_vector* function is designed for debugging, it can print an input vector as a string series. *format* function is a poly-fill for *std::format*[2] as *clang++14* marked this feature as incomplete[4] (*clang++15* does not mention this feature, neither).

1.2 psrs

This *namespace* is used for exporting the public interface function, *psrs*, which is exactly the implementation of the PSRS paper, and encapsulating other functions which are only used by *psrs*. The function *psrs* takes a reference of an integer vector, the number of threads, and a optional reference of an 64-bit integer vector to store runtime of each phase, and returns an sorted integer vector as the result, where the original vector remains unchanged. The following pseudo-code 1 shows the workflow of *psrs* function. The workflow of *parallel_sort* function in the *psrs* function is described in the original paper, which contains four phases. Moreover, for measuring the total time in details, here I define two more phases, where Phase 0 is the initialization phase and Phase 5 is the collection and joining phase. The initialization phase generates payloads for each thread as well as some global variables stored in the *Globals* class. The collection phase concatenates the array from each thread, and joins the threads.

```
Algorithm 1: PSRS Workflow
input  : An unsorted array $V$ of size $s$; the number of threads $t$
output: A sorted array $V^{'}$
1  pthreads payloads $payloads \longleftarrow \phi$
2  pthreads identifiers $pthreads \longleftarrow \phi$
3  for $i \leftarrow 0$ to $t - 1$ do
4      initialize($V$, $payloads[i]$)
5  for $i \leftarrow 1$ to $t - 1$ do
6      pthread_create($pthreads[i]$, parallel_sort, $payloads[i]$)
7  parallel_sort((void*)&payloads[0])
8  result array $V^{'} \longleftarrow payloads[0].result$
9  for $i \leftarrow 1$ to $t - 1$ do
10     $V^{'} = V^{'} \cup payloads[i].result$
11     pthread_join($pthreads[i]$)
12 return $V^{'}$
```

2 Experiments

In this section, I provide details of the setup of the experiments, which is used for evaluation the implementation of *PSRS*.

2.1 Platform

Here is the spec sheets of the platform for testing.

- CPU: Intel Core i7-12800H, 6 Performance Cores + 8 Efficiency Cores, 12 + 8 Threads (with Hyper-Threading enabled)
- OS: Windows 11 Enterprise 22H2 with WSL2 (Ubuntu 22.04, kernel: 5.15.57.1-microsoft-standard-WSL2)
- Compiler: Clang++15 with CXX standard of C++23

2.2 Setup

The sizes of arrays are 32M, 64M, 128M, 192M, 256M, 320M, where the contents are randomly generated by *std::uniform_int_distribution* function ranging from *INT32_MIN* to *INT32_MAX* in uniform distribution. The numbers of threads are even numbers between 2 to 20. Each size runs 10 times and the result is the average of the last 5 runs. At the start of each run, the main function generates an vector of a certain size containing uniformly distributed random integers, and calls sequential sorting function on a copy of the vector, as the built-in sorting is in-place, then calls *psrs* in a for-loop of different numbers of threads. Thus, In each run, the random generated data is the same for both sequential and parallel sorting functions. Because *psrs* uses barrier synchronization, the elapsed time for each phase is the longest time among all threads, and the sum of the elapsed times of all phases is the total elapsed time of *psrs*. Therefore, every thread records and reports its own elapsed time of every phase, and the main thread calculates the actual elapsed time of each phase. At last, the correctness can be validated by comparing the copy sorted by the built-in sequential function and the data returned by *psrs*.

3 Results

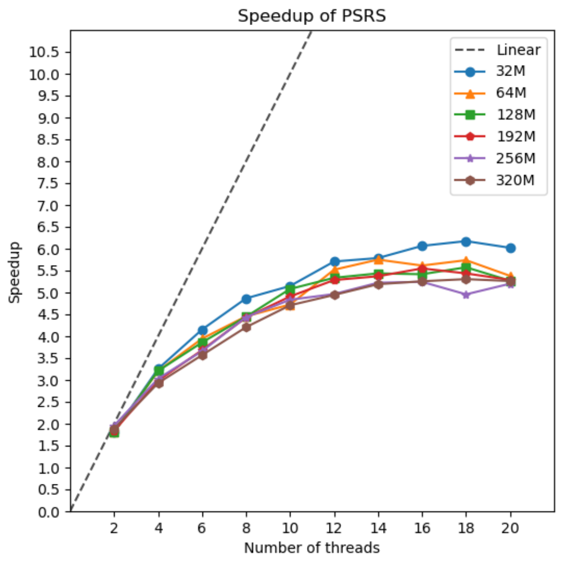

Figure 1: Speedup of PSRS

sizes/threads	Sorting Times (in seconds)										
	1	2	4	6	8	10	12	14	16	18	20
32M	2.263	1.252	0.695	0.545	0.465	0.440	0.397	0.391	0.373	0.367	0.376
64M	4.890	2.697	1.526	1.239	1.098	1.035	0.886	0.85	0.871	0.852	0.909
128M	9.963	5.508	3.106	2.582	2.241	1.961	1.867	1.833	1.839	1.787	1.891
192M	16.289	8.867	5.458	4.422	3.678	3.318	3.083	3.033	2.935	2.997	3.081
256M	22.303	11.405	7.362	6.093	5.025	4.614	4.493	4.27	4.254	4.499	4.289
320M	28.101	14.894	9.593	7.873	6.679	5.969	5.684	5.415	5.346	5.297	5.341

Table 1: Sorting Times

Figure 1 is the speedup graph and Table 1 is the corresponding elapsed time of each size with certain number of threads. Moreover, Table 2 shows the distribution of elapsed time in each phase with different number of threads, where the size of the array to sort is 320M. It is obvious that the maximum speedup is 5.751 with 14 threads at 64M, and the implementation slows down for more than 16 threads. More importantly, the diminishing returns happen after more than 6 threads.
From the breakdown of execution times according to phases, the elapsed time for Phase 1 decreased significantly from 2 threads till 6 threads, and for Phase 2 is barely noticeable. This is also the reason why I choose microsecond to be the unit for Table 2 instead of millisecond or seconds like in Table 1. However, it does increase as the number of threads increases as expected. Moreover, for other phases, the execution time increases as the number of threads increases. Although Phase 0 and 5 defined in Section 1.2 are shorter comparing to Phase 1 and 4, they both contribute a noteworthy amount of execution time.

Phase/threads	Sorting Times (in microseconds)									
	2	4	6	8	10	12	14	16	18	20
0	290379	195223	211537	213171	219143	216593	215966	218809	216825	226122
1	13067073	8102512	6392375	5142978	4348529	4008446	3663934	3516166	3346533	3336392
2	1	2	2	3	4	6	8	12	15	20
3	183521	136226	130268	138662	142625	148758	153677	143449	143480	139049
4	1075799	840668	768302	766295	846212	896549	936764	984962	1061393	1130113
5	277203	317938	370255	417930	412609	413803	444231	482446	528577	508934
Total	14893976	9592569	7872739	6679039	5969122	5684155	5414580	5345844	5296823	5340630

Table 2: Sorting Times in Each phase (size = 320M)

4 Analysis

- Phase 1 is where local data is sorted. Since the time complexity of comparison sorting is at least $\mathcal{O}(n \log n)$, the only way to reducing the run-time is reduce the amount of data for each threads, which is why increasing the number of threads leads to decrement of execution time as long as one physical core execute only one thread. Context switching creates a huge amount of overheads. Phase 4 is similar, the workload of merging is reduced by increasing the number of threads.

- Phase 2 is purely sequential and it will become to the bottleneck when the number of threads increases according to Amdahl's Law. However, in this experiment, the number of threads is too small to make it to be the bottleneck, and the increasing trend may also be affected by non-determinism.

- The main propose of Phase 3 is partitioning the local data and distributing the partitions, which mainly depends on the number of threads. Moreover, Phase 0 and Phase 5 also depends on the number of threads only. Therefore, the execution times of these phases have the same trend. They may be affected by non-determinism as well, due to the relatively short execution time.

5 Limitations

- *pthread* vs *std::thread* (proposed in *C++11*)
  The API and implementation of pthread (POSIX thread) are generally available on many UNIX/UNIX-like POSIX-conformant operating systems. However, it is not available on Windows directly[8]. Windows has its own multi-threading API. *C++* standard library introduced *std::thread* since *C++11*. It provides a higher abstraction of multi-threading APIs, which is cross-platform. The implementation of *PSRS* in this report uses *pthread* only. In order to compile and run tests on Windows, one option is rewriting it by using *std::thread*.

- Heterogeneous CPU Topology
  Heterogeneous CPU topology is a system design that all cores use the same instruction-set architectures, but the speed of cores vary[7]. Intel produced its 12th generation CPU in this design to provide better power efficiency. Although the main purpose of regular sampling in *PSRS* is balancing the load between threads, scheduling the same amount of data between performance-cores and efficiency-cores is not actually balancing. Therefore, the results of the experiments may be affected by the different computation powers between different cores.

References

[1] N. Athanasiou. Easily measure elapsed time, February 2014. URL `https://stackoverflow.com/a/21995693`. Last modified at Jan 2022.

[2] cppreference.com. formatting library and std::format, 2022. URL `https://en.cppreference.com/w/cpp/utility/format/format`.

[3] cppreference.com. std::chrono::steady_clock, 2022. URL `https://en.cppreference.com/w/cpp/chrono/steady_clock`.

[4] eerorika. Can't use std::format in c++20, April 2022. URL `https://stackoverflow.com/a/71778006`.

[5] iFreilicht. std::string formatting like sprintf, October 2014. URL `https://stackoverflow.com/a/26221725`. Last modified at March 2022.

[6] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Comput.*, 19:1079–1103, 1993.

[7] wikipidia. Heterogeneous computing, September 2022. URL `https://en.wikipedia.org/wiki/Heterogeneous_computing`.

[8] wikipidia. pthreads, 2022. URL `https://en.wikipedia.org/wiki/Pthreads`.

[9] Zorawar. How do i print out the contents of a vector?, May 2012. URL `https://stackoverflow.com/a/10758845`. Last modified at May 2022.