Algorithm Engineering
WS 22/23

Sanket Datta
Toru Tsukamoto

Parallel Sorting

Abstract

We implement and compare the practical performance of two different variants of two well known sorting algorithms,namely, Classic merge sort, Parallel merge sort and Classical quick sort and parallel quick sort algorithm. We first explain the basic algorithms,then test them with different size of data and finally compare the performance of each of them. We test both methods with different sizes of randomly generated numbers. Our tests demonstrate that, regardless of the quantity of the input data, parallelization elongates computation times. Given that Python does not support enough parallelization, this is to be expected.

1 Introduction

In real world sorting data is a very big part of life where tech giant strive to sort billions of data every moment for getting various insights from the data. As time progressed and data became larger, New sorting methods came up to tackle with the enormous amount of data. Numerous methods, including bubble sort, insertion sort, quick sort, merge sort, etc., have been developed for efficiently processing enormous volumes of data.
The classic merge sort divides the list of numbers into equal halves until each list has just one element and then merges it from the bottom up. Quick sort is a different sorting method that divides the supplied array around a pivot element that is chosen as a starting point for the sort. All items that are less than the pivot are placed before the pivot, and all elements that are larger than the pivot are placed after the pivot. The pivot element is chosen to be the first item on the list in this experiment.
On the other hand, a large number of CPUs, on the other hand, have been added to the majority of computers in recent years. Even a standard personal computer has several to several dozen CPUs, and top-tier supercomputers frequently have several thousand or more cores. In order to increase their efficiency, sorting algorithms could theoretically be run in parallel.
In this report we extend Merge sort and Quick sort to run in parallel. Then, we compare two sorting algorithms, merge sort and quick sort with their parallel variant with a focus of their practical performance with different kind of data loads.
In a parallel merge sort, the list of numbers is divided into smaller chunks so that it takes less time than the traditional approach, and then the chunks are finally merged bottom up so that each thread takes up a piece of the list or each number to locate its place in the sorted list. Quick sort involves recursively re-sorting the data into two groups, large and small, while Parallel quick sort executes that both this rearranging and recursion are done in parallel.
The remainder of this work is structured as follows.Section 2 will explain the necessary background information regarding the experiment. Section 3 will explain the two algorithms, provides information about the used algorithm engineering techniques and gives some implementation details. In Section 4, we describe the experimental setup and provide empirically results. In addition, we discuss the Data and Hardwares used for this experiment. Then we compare the stats using data visualization. Finally, in section5, we discuss the outcomes.

2 Preliminaries

The two stages of mergesort are as follows. Divide and conquer is the key concept in it. The input is split into two equal-sized sublists in the first step, recursively, until each list has only one entry. By con-

tinually comparing the two initial components and adding the smaller one to a result list, every two successive sublists are combined into a bigger sorted list in the second phase. Up till the whole array is sorted non-decreasingly, the merging process is repeated on the bigger and larger sorted sublists.
The Quick sort is also contains two steps based on the divide and conquer idea. The pivot element is decided upon. The numbers on the left of the pivot element are smaller than it, while the numbers on the right are larger than it. This is how the list of numbers is split and sorted. Until a sorted list of integers is formed, this is performed recursively on the side sublists that were established by the pivot element.

3 Algorithm & Implementation

This section provides information about the actually used algorithms and their respective implementations. It should roughly cover the following four topics:

- Classical Merge Sort:

We implemented a recursive top-down merge sort instead of a bottom up merge sort that does not use recursion.
At first, we split the input S into two nearly equal large lists, Sa = x1, x2,..., x(n/2) and Sb = x(n/2+1),....xn, we recursively execute the partitioning on each of the sublists. If there is just one entry in the each list, the task is stopped.
Then, we repeatedly combine the two sublists into one big sorted list. To accomplish this, take the smallest number of each sublists' leftmost components. This minimum is then appended to the result list, until both sublists are exhausted. Since the sublists were sorted non decreasingly, the result is sorted as well.
The complexity of this algorithm is O(n log n) and its space consumption is O(n).

- Parallel Merge Sort:

The sections of the list of numbers are split into two lists and then recursively combined until there are only single items. Each thread handles one segment. Then we start combining them, starting with two sublists. The position of an element in the sorted list is determined by taking its index from the first sublist, its position from the second sublist, and the sum of the two. For each segment of the list that has been sorted, this is repeated from the bottom up. To save time, we switch to the conventional merging if the machine runs out of cores.
The time consumption is logn for the depth of the merge sort and for merging it takes (logn)2 considering the number of cores is same as the number of inputs. The space consumption is O(n). Hence the parallelization depends on the number of cores provided. However, As we tested in python we found that though theoritically the algorithm is supposed to work faster than classic merge sort but the threadpool executor package in python is slow to execute as the thread creation and closure in the end takes some time to process. Thus it takes more time for execution than the classic merge sort.

- Classical Quick Sort:

The fundamental principle of the traditional quick sort is to pick a pivot reference number and repeatedly divide it into two groups, large and small. Ideally, n data are split into two equal parts each time for sufficiently random data, yielding a division of log 2n depth. The theoretical time-computation complexity is O(nlogn) because each data item is reordered only once at each depth.
Recursion was used during implementation to carry out the iterations. A Python list was used to represent the data, and the data with the smallest subscript served as the pivot. The original data in memory was kept on the memory while rearranging one by one by using a new memory area for saving one data.

- Parallel Quick Sort:

We parallelised quick sort recursion and rearrange.
Parallelisation of recursion is actually only done the first very limited number of times, as the number of cores increases exponentially. On the other hand, rearrangement of a data chunk based on a particular pivot can be done in parallel each time. The data is divided into several even smaller chunks and compared in parallel with the pivot. At this point, the data sorted by chunk needs to be combined into one. Since extra computation is required for this process, the theoretical time-computation complexity of

p-parallel quick sort is O((n/p)(logn) + (logp)(logn)). In this experiment, as we had 8 CPUs available to us, we decided to perform the first recursion in parallel and rearrange on 4 cores. Due to the overhead of parallelisation, rearrange was only parallelised when the chunk length was greater than 32.

4 Experimental Evaluation

4.1 Data and Hardware

Randomly arranged integer data with no duplicates is used as input data. The number of data can be 100, 1000, 10000 or 100000.
Table 1 shows the specifications of the computers used in the experiment.

Table 1: Computer Specifications

Chip	Apple M1 Pro
Core	8 CPU, 14GPU, 16 Neural Engine
Memory	LPDDR5 16GB
Storage	APPLE SSD

4.2 Results

Fig 1 shows the time taken to sort data of four different sizes using four different sorting methods: classic merge sort, parallel merge sort, classic quick sort and parallel merge sort. In this experiment, the computation was faster without parallelisation than with parallelisation. In addition, quick sort was faster when not parallelised and merge sort was faster when parallelised. Note that when parallelised, it took several minutes of computation time for one sort with 100 000 data, and it was difficult to use larger data. In addition, for all sorting methods, the increase in computation time is clearly smaller than O(n2) when the number of data increases.

5 Discussion and Conclusion

Our experiments show, that for Mergesort as well as Quicksort the parallelization can lead to a significant

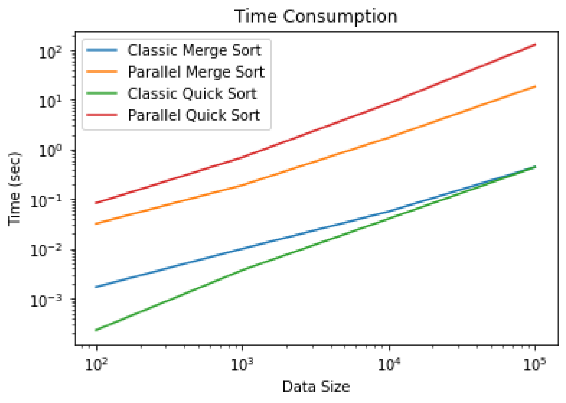

Figure 1: Time Consumption in Sorting Various Number of Data

decrease of the running time theoritically but choosing python for parallelization does not help decreasing the time. One reason for this is that python has the Global Interpreter Lock(GIL), which does not allow users to freely parallelise using threads. Though it has support of libraries which help running multiple threads at a time but starting the thread and then vacating the thread for future use takes some time of action which in the end delays the program's running time.
In parallel quick sort, instead of thread parallelism, process parallelism was used. As the data could not be shared, all the data from each process had to be passed on, and the data merged after the parallel process had finished. This made the computation even slower.
Hence In our experiment however, Classical version of merge sort and quick sort produces better result than the parallel version.