Parallel Quicksort with OpenMP

Mingyang Liu, Rui Lyu

April 17, 2024

Abstract

Sorting is a common and fundamental problem in the field of computer science. Improving the speed of sorting is a valuable pursuit. In this report, we propose a way to paralle quicksort using OpenMP. The main method we use to improve the performance is setting cut-offs. Our work reveals the importance of cache efficiency and task scheduling regarding the performance of parallel program.

1 Introduction

Sorting plays a crucial role in many applications across diverse fields. Nowadays, as datasets continue to grow at a rapid pace, it is important for us to find faster sorting algorithms to handle larger datasets. Meanwhile, the rapid advancement of multicore processors and parallel computing technologies provides us a way to achieve this goal by paralleling existing sorting algorithms. By choosing the proper strategy of parallelizing, we can significantly enhance the performance and scalability of the sequential sorting algorithm.

In this report we parallelize quicksort, an efficient and well-known sorting algorithm. We start by exploring how others handle similar challenges in parallelizing applications. Next, we explain our idea and how we tackle problems as we work towards the solution in detail. The inherent mechanism of quicksort often leads to unbalanced partition of the input data, causing uneven workloads among threads. Our approach focus on achieving a more balanced workload distribution among threads and reducing overheads of parallelism. Finally, we compare our algorithm with sequential quicksort and another parallelized quicksort to analyze the performance of our implementation.

2 Literature Survey

To improve the performance of parallel quicksort, we need to address some common problems in parallelizing applications. In order to develop a parallel quicksort with great performance, critical considerations include load balancing, optimizing cache efficiency, and minimizing overhead of parallelism. There are many researches related to how to deal with these challenges, so we can leverage on their findings to improve our solution.

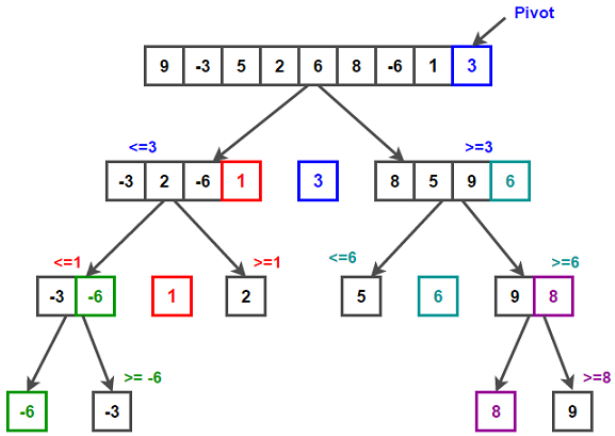

Figure 1: Quicksort

2.1 A High Level Pattern

Quicksort is a sorting algorithm that follows the divide-and-conquer pattern 1. The research in [3] proposes a template of parallelizing programs that apply the divide-and-conquer strategy. It gives us a general idea of how to separate quicksort into different phases and apply OpenMP directives to each section. The drawback of this research is that it does not provide any specific implementations of the algorithm. Also, it does not propose methods of how to improve the performance of divede-and-conquer programs. Based on part of the idea of it [3], we implement a parallel quicksort and provide ways to enhance the performance of parallel programs.

2.2 Task Granularity Control

The overhead associated with creating and managing tasks is a key factor that influences the performance of parallel programs. Reducing the overheads of creating tasks [4] is important if we want to ensure the overall performance of the parallel program. Therefore, we need to control task granularity and schedule tasks carefully. This work [4] explores several scheduling strategies of OpenMP tasks. In another research, the author introduced multiple cut-off approaches and evaluate their performance[5]. The cut-off strategy decides whether new tasks should be spawn or execute sequentially. The good part of this research is that it provides a detailed analysis of several cut-off strategies. However, both researches mentioned above focus on theoretical evaluation rather than implementation. They lack concrete guidance on which strategy to choose for different types of applications. Our work provide specific usage of several strategies to improve the performance of the parallel program.

2.3 Load Balancing

Imbalanced workload among threads is another problem that may lead to bad performance of parallel programs. In this research, the author uses Unbalanced Tree Search as benchmark to analyze several load balancing strategies [8]. For example, work aggregation and the cut-off strategy mentioned above. While the author states work aggregation should be used for efficient load balancing, they consider cut-offs unsuitable. However, in the context of parallel quicksort, by setting the threshold based on the array size, the cut-off strategy can achieve load balancing and thus improve the performance. For the case of quicksort, another way to achieve a better load balance is finding a proper pivot. The median of medians algorithm [2] is a good choice to find a suitable pivot.

2.4 Cache Efficiency

We believe that an effective strategy to improve cache efficiency is partitioning the data into proper size that can fit into the cache. This research [7] introduces some memory optimization techniques of several sequential sorting algorithms and provides performance analysis. This work supports our assumption of partitioning the data. While it focus on sequential program, we implements memory optimization for parallel program.

3 Proposed Idea

We are using OpenMp, a widely used API that supports shared-memory parallel programming, to parallelize quicksort. The objective of our project is to make our program outperform the sequential quicksort and some other parallel quicksort in terms of the speed.

3.1 Directives

We use OpenMP Tasks directives rather than Sections for the parallelizing. Compares to OpenMP Sections, Tasks are more flexible. We can dynamically create and schedule tasks based on our program, resulting in better performance of the program.

3.2 Pivot Selection

The choice of pivot tightly relates to the load balance and can impact the speed of quicksort. If the partition is extremely unbalanced, meaning one partition is much larger than another one, the runtime of quicksort could degrade to $O(n^2)$. To address this problem, we decide to apply the median of medians algorithm for pivot selection. Although choosing a random pivot can also reduce the possibility of the worst case run time, the median of medians algorithm offers a more stable pivot. Median of 3 and median of 5 are both widely used strategy and do not differ a lot in terms of performance, and we choose median of 3

for our pivot selection.

3.3 Quicksort Implementation

To implement the sequential quicksort recursively or iteratively is another choice we need to make. We decide to use recursive quicksort in our program since it follows the divide-and-conquer strategy naturally, making the code simple and clear. Additionally, the iterative quicksort requires the using of stack, so it does not provide obvious memory savings compares to the recursive quicksort.

3.4 Cut-off Strategy

Regarding the problem of load balance and overhead, we use the cut-off strategy to perform task scheduling. In our program, we have four different threshold serve as cut-off points.

3.4.1 Cache Size

We use threshold *cache_size* that is set to match the L2 cache size to improve the cache efficiency. If current array size is greater than the cache size, we keep partitioning the array and create new tasks with smaller array size. This threshold is used to ensure that only tasks with array size smaller than the cache size can be execute. If the array size falls below the threshold, it can reside entirely in the cache, leading to fewer cache misses and improved cache efficiency. Although L1 cache is faster, it is too small. After testing, we find that using L1 cache size does not provide enhancement of the performance, so we decide to stick with L2 cache.

3.4.2 Load Based Cut-off

we have threshold $cutoff$. Tasks with array size that is smaller than this threshold do not spawn new tasks since the overhead of creating and managing new tasks could exceed the benefits. The value of $cutoff$ depends on the size of the input array. For our work which sorts array with 10,000,000 and 100,000,000 elements, we test 100, 1000, 10000 and 10,0000. In most cases, setting $cutoff$ to 1,000 provides the best performance. In addition, if the array size is less than 16, then sequential insertion sort is used since it performs better than quicksort for small array.

3.4.3 Task Number and Recursion Depth

We have threshold $task\_num$ and $depth$ to prevent excessive task creation. We set $task\_num$ to two times the number of threads. If total number of tasks exceeds $task\_num$, new task

will not be created. Given our limited number of threads, tasks cannot be executed immediately. Consequently, if the number of tasks exceeds the number of available threads a lot, many of them are left waiting for execution, leading to significant overhead.

Similarly, if the recursion level exceeds *depth*, task creation is halted. We set *depth* to be slightly greater than $\log(n)$, where n is the array size. A recursion level exceeding *depth* indicates a highly imbalanced partition and we need to stop producing new tasks.

For instance, suppose an array with size n is always partitioned into 1 element and $n-1$ elements. In this case, instead of constantly generating new tasks with small size, sorting the array directly could be faster since the excessive tasks created could lead to high overhead.

Also, since we handle the left subarray before the rigth subarray in our program, it is possible that left subarray keeps dividing itself and spawning new tasks. Therefore, the right subarray is blocked and cannot be processed. We can solve this issue with *depth* since it limits the recursion level of the left subarray.

3.4.4 Lock Variable

We have a variable *lock* to control the task creation. Essentially, *lock* decides when to use thresholds other than *cache_size*. Initially, *lock* is set to 0. Instead of executing existed tasks, each thread is partitioning the array and creating smaller tasks when the program starts running. Threads keep generating new tasks until all tasks are around cache size so the data can fit into the cache. Since *lock* is still 0, threads do not produce new tasks but start to execute existing tasks. *lock* is set to 1 when at least one task with cache size finishes execution. Then, thresholds other than *cache_size* are used to manage the task creation. In this way, we can minimize the possibility of generating or executing too many small tasks before the tasks with cache size are depleted. Therefore, we reduce the overhead of creating and managing tasks by limiting the number of tasks. Also, we improve cache efficiency by preventing frequent switching between small tasks.

3.4.5 Threshold Selection

To improve the performance of our program by using cut-off strategies, we incorporating thresholds in the program gradually. Initially, We have the program without any threshold, then we introduce new cut-off strategy iteratively, evaluating their impact on the performance of the program. If new added cut-off strategy enhance the performance, we integrate it into the program. Otherwise, we discard it so that only useful cut-off is adopted. The test of thresholds is shown in Table 1

Table 1: Threshold Test with 100,000,000 elements

Threads	base	depth	cache_size + lock + task_num	cache_size + lock + task_num + depth
8	7.309	6.473	6.349	6.257
16	5.495	4.869	4.712	4.672

4 Experimental Setup

4.1 Hardware

The results in this report are based on NYU Courant Crunchy server that uses four AMD Opteron 6272 CPU. The specification of the machine is provided in Table 2 and the specification of the CPU is provided in Table 3.

Table 2: Machine Specification

CPU	Four AMD Opteron 6272 (2.1 GHz) (64 cores)
Memory	256GB
OS	CentOS 7

Table 3: CPU Specification

Model name	AMD Opteron(TM) Processor 6272
Architecture	x86_64
CPU(s)	64
Thread(s) per core	2
Core(s) per socket	8
Socket(s)	4
CPU MHz	2100.119
L1d Cache	16K
L1i Cache	64K
L2 Cache	2048K
L3 Cache	6144K

4.2 Benchmark

We use the random generated array of integers as our benchmark.The size of the array is 10,000,000 and 100,000,000.

5 Results and Analysis

We test the performance of our algorithm and the sequential quicksort. Also, we compare the performance of our algorithm with parallel quicksort that is implemented by others. The parallel quicksort we compared to is adapted from [1] and [6]. To measure the speedup over sequential code and other's parallel code, we use array of size 10,000,000 and 100,000,000 since array with size less than 1,000,000 is too small to show obvious effect of parallelism. For all results, we use **pqsort** to refer our implementation of parallel quicksort, and **pqsort other** to refer other's implementation. **speedup I** is the speed up over sequential sort,

and **speedup II** is the speed up over other's parallel quicksort. Running time is measured in seconds.

Table 4: Array with 10,000,000 Elements

number of threads	pqsort	sequential	speedup I	pqsort other	speedup II
2	1.806	3.814	2.112	2.188	1.212
4	0.994	3.814	3.837	1.186	1.193
8	0.601	3.814	6.346	0.727	1.210
16	0.423	3.814	9.017	0.491	1.161

Table 5: Array with 100,000,000 Elements

number of threads	pqsort	sequential	speedup I	pqsort other	speedup II
2	20.329	43.535	2.142	23.671	1.164
4	10.633	43.535	4.094	12.367	1.163
8	6.02	43.535	7.232	7.068	1.174
16	4.493	43.535	9.689	5.363	1.193

Table 4 shows the performance of different algorithms of sorting array with 10,000,000 elements. Table 5 shows the performance of different algorithms of sorting array with 100,000,000 elements.

It is obvious that our algorithm performs much better than sequential quicksort. The speedup increases as we increase the number of threads. This is reasonable because more threads are working and the loss of the parallelism is less than the gain from it.

In some cases such as having 2 threads, we have efficiency over 1. Cache efficiency could be the reason. With single core, the entire array often exceeds the cache size so it cannot reside on the cache, leading to frequent cache misses and increased data movement. However, with more cores and threads, the array is partitioned into smaller segments that can fit in each cache, so there are less data movement and cache misses, resulting in enhanced cache efficiency and overall performance. As we increases the number of threads, the efficiency may decrease. Since more threads often leads to more cost of parallelism, the growing overhead can potentially outweighs the benefits gained from cache efficiency and other improvement, leading to lower efficiency.

The results above indicate that our parallel quicksort outperforms that algorithm of others. Comparing to that parallel code, our work has better cache efficiency and task creation control.

In addition, it is safe to say that our algorithm has great scalability. As presented in **speedup I**, with fixed number of threads, we see speedup as the problem size increases.

With fixed problem size, we can see speedup as the number of threads increases.

6 Conclusion

- Our algorithm performs better than the sequential quicksort and a parallel quicksort implemented by others.
- Cache efficiency is critical for good parallel program performance, and it can even lead to the efficiency that exceeds 1.
- Proper task scheduling such as setting cut-offs can reduce the overhead and balance the workload distribution, resulting in a better performance.

References

[1] Sinan AL-Dabbagh. Parallel Quicksort Algorithm using OpenMP. 7 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. A divide-and-conquer parallel pattern implementation for multicores. In *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems*, SEPS 2016, page 10–19, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of openmp task scheduling strategies. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *OpenMP in a New Era of Parallelism*, pages 100–110, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[5] Alcides Fonseca and Bruno Cabral. Evaluation of runtime cut-off approaches for parallel programs. In Inês Dutra, Rui Camacho, Jorge Barbosa, and Osni Marques, editors, *High Performance Computing for Computational Science – VECPAR 2016*, pages 121–134, Cham, 2017. Springer International Publishing.

[6] Kil Jae Kim, Seong Jin Cho, and Jae-Wook Jeon. Parallel quick sort algorithms analysis using openmp 3.0 in embedded system. In *2011 11th International Conference on Control, Automation and Systems*, pages 757–761, 2011.

[7] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, 1999.

[8] Stephen Olivier and Jan Prins. Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, 38:341–360, 10 2010.