PARALLEL BUCKET SORTING ALGORITHM

Hiep Hong
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000

hiepst@yahoo.com

ABSTRACT

Bucket sorting algorithm achieves O(n) running time complexity in average. It is very fast compared to any comparison-based sorting algorithms that usually have a lower bound of Ω(n log n). Given an unsorted array of n positive integers, bucket sorting algorithm works by distributing the elements into m ordered buckets each contains zero or more elements. Each bucket is then sorted using either a different sorting algorithm or recursively using bucket sorting algorithm. At the end, elements are gathered from each bucket in order, so the input array is sorted. With the O(n) complexity, the bucket sorting algorithm can execute and finish sorting very quickly in sequential processing. However, this execution time can even be improved by parallelizing the sorting of each bucket. This paper will show how the algorithm can be converted to a parallel algorithm and be implemented and executed using OpenMP API. Furthermore, it is a scalable algorithm as its performance can be linearly improved with the increase of core number of multicore processors. Experiment results show the efficiency of the algorithm.

1. INTRODUCTION

Sorting algorithm can be classified into two categories, comparison-based sorting and non comparison-based sorting. For comparison-based sorting, the best-case performance is Ω(n log n) while the worst case performance is O(n^2). The non comparison-based sorting is different from the comparison-based such that it depends only on the key and address calculation. When the values of keys are finite ranging from 1 to m, the computational complexity of non-comparison-based sorting is O(m+n). Particularly, when m=O(n), the sorting time can reach O(n) [8].

Bucket sort is an example of a sorting algorithm that, under certain assumptions on the uniform distribution of the input, breaks the lower bound of Ω(n log n) for standard comparison-based sorting [3]. For example, suppose that there is a set of $n = 2^m$ integer elements to be sorted and that each is chosen independently and uniformly at random from the range $[0, 2^k)$, where $k \geq m$. Using bucket sort, the set can be sorted in expected time O(n). Since bucket sort is a completely deterministic algorithm, the expectation is over the choice of the random input.

Bucket sort works in two phases. In the first phase, we place the elements into n buckets where the j^{th} bucket holds all elements whose first m binary digits correspond to the number j. For instance, if $n = 2^{10}$, bucket 3 contains all elements whose first 10 binary digits are 0000000011. When $j < l$, the elements of the j^{th} bucket all come before the elements in the l^{th} bucket in the sorted order. Assuming the each element can be placed in the appropriate bucket in O(1) time, the phase requires only O(n) time. Since it is assumed that the elements to be sorted are chosen uniformly, the number of elements that fall into a specific bucket follows a binomial distribution B(n, 1/n).

In the second phase, each bucket is sorted using any standard quadratic time sorting algorithm such as Quick sort or Insertion sort or recursively using the same Bucket sort. Each bucket is concatenated in order produces the sorted order for the elements. Under the uniform distribution of the input, bucket sort falls naturally into the balls and bins model in which the elements are balls, buckets are bin, and each ball falls uniformly at random into a bin. This follows a binomial random variable B(n, 1/n) whose expectation is n. Therefore, the expected time spent in the second phase of bucket sort is only O(n).

Utilizing the evolving multicore processor architecture, this paper shows how the bucket sorting algorithm can be converted to a parallel algorithm that is implemented and executed using OpenMP API. Furthermore, parallel bucket sort is a scalable algorithm as its performance can be improved with the increased number of processor cores. Experiment results show the efficiency of the algorithm. As a result, the work in this paper contributes to the utilization of multicore architecture by converting sequential sorting algorithm to parallel sorting algorithm while preserving the integrity of it.

This paper is divided in to six sections. Section 2 discuses about related work on parallel sorting algorithms. Section 3 introduces buck sort in sequential and parallel versions. Section 4 provides the introduction to OpenMP API. Section 5 discusses the experiment results and performance analysis. Section 6 concludes the paper and provides some future work in consideration.

2. RELATED WORK

Zhao and Min try to improve bucket sorting's efficiency by focusing on the uniform distribution of records into the buckets. In their hypotheses, the data collected from empirical studies usually follow a certain probability distribution in a certain interval. To sort this kind of data, their paper proposed an innovative method through constructing a hash function based on its probability density function. Then n records can be allocated into n buckets uniformly according to the value of their key, which guarantees the sorting time of the proposed bucket sorting algorithm to achieve O(n) under any circumstances [9].

Kim et al.'s work presents the effectiveness of parallel programs in embedded hardware architecture using OpenMP 3.0. They point out that increasing CPU clock speed causes problems, such as rising temperature or power consumption, which is also known as "heat wall" or "power wall", respectively. These problems may be solved by multi-core architecture, MIMD machine, which is either has shared or distributed memory model. While OpenMP and Pthreads API are commonly used to develop parallel programs for the shared-memory model, OpenMPI and PVM (Parallel Virtual Machine) are commonly used in distributed memory model in which each core has its own memory and transfers data by the MPI (Message Passing Interface) protocol. They parallelize by breaking the programs into many pieces each can be handled by a thread. However, it is impossible to divide a sequential program into the ideally independent components because some shared memory problems or subordinate task flows are inevitable to make a parallel program. In addition, creating too many threads, regardless of the number of cores, can seriously degrade system performance, because of overheads from starting and terminating threads and the race condition for occupying limited embedded hardware resource. Limitation of cache memory and data sharing architecture induce frequent data movement overhead between cache and main memory. These problems are important in embedded systems, since they have strict limitations in the number of cores and memory size. Therefore, when they develop a parallel program at the embedded level, they have to consider the limitations of the hardware architecture. In their experiments, they parallelize many different versions of quick sort algorithm using OpenMP API. The first version is quick sort using task decomposition. The second version is hybrid quick sort using load balancing where each sub arrays size is designed to fit in cache size in order to make cache friendly usage. The third version is introspective quick sort using insertion sort to avoid worst case of quick sort by limiting the depth of partitioning. This version of the algorithm is more cache friendly than hybrid quick sort algorithm because it divides initial big problem into small sub problems, which means cache handles smaller and smaller data as the progress goes on, and there is no need to make buffer memory to use insertion sort. Among the three versions, introspective quick sort performs best in the embedded environment. They analyze the performance effectiveness by considering the number of threads and cache memory size and by using the GNU Gprof profiler to measure run time.

3. ALGORITHMS

3.1 Sequential Bucket Sort

Bucket sort is an example of a sorting algorithm that, under certain assumptions on the uniform distribution of the input, breaks the lower bound of $\Omega(n \log n)$ for standard comparison-based sorting [3]. For example, suppose that there is a set of $n = 2^m$ integer elements to be sorted and that each is chosen independently and uniformly at random from the range $[0, 2^k)$, where $k \geq m$. Using bucket sort, the set can be sorted in expected time O(n). Since bucket sort is a completely deterministic algorithm, the expectation is over the choice of the random input.

Algorithm 1 Sequential Bucket Sorting Algorithm.

```
1: Initialize an array of empty "buckets".
2: Go over the unsorted original array and put elements in the bucket based on their values.
3: Sort each non-empty buckets.
4: Visit the array of buckets in order and put the sorted elements back to the original array.
```

Figure 1. Sequential Bucket Sort Algorithm

Bucket sort works in two phases. In the first phase, we place the elements into n buckets where the j^{th} bucket holds all elements whose first m binary digits correspond to the number j. For instance, if $n = 2^{10}$, bucket 3 contains all elements whose first 10 binary digits are 0000000011. When $j < \ell$, the elements of the j^{th} bucket all come before the elements in the ℓ^{th} bucket in the sorted order. Assuming the each element can be placed in the appropriate bucket in O(1) time, the phase requires only O(n) time. Since it is assumed that the elements to be sorted are chosen uniformly, the number of elements that fall into a specific bucket follows a binomial distribution B(n, 1/n).

In the second phase, each bucket is sorted using any standard quadratic time sorting algorithm such as Quick sort and Insertion sort or recursively using bucket sort. Each bucket is concatenated in order produces the sorted order for the elements. Under the uniform distribution of the input, bucket sort falls naturally into the balls and bins model in which the elements are balls, buckets are bin, and each ball falls uniformly at random into a bin. This follows a binomial random variable B(n, 1/n) whose expectation is n. Therefore, the expected time spent in the second phase of bucket sort is only O(n).

Figure 1 shows the sequential version of bucket sorting algorithm.

3.2 Parallel Bucket Sort

In the parallel version of bucket sorting algorithm, k threads are used to perform the sorting. The original array of N elements is partitioned into k sets each has N/k elements. This is preparation step before each thread is assigned to put N/k elements in the buckets based on their values. After all the elements have been put in the M buckets, the next step is partitioning the buckets into k sets each has M/k buckets. Now each thread is assigned to sort M/k buckets by using any standard quadratic time sorting algorithm such as Quick sort and Insertion sort or recursively using bucket sort.

The pseudo code of the algorithm is shown in figure 2.

```
Algorithm 2 Parallel Bucket Sorting Algorithm.
N: number of unsorted elements
M: number of buckets, k: number of threads
1: Initialize an array of empty “buckets”.
2: Partition the original array into k sets, each has N/k elements.
3: In parallel, each thread performs the following:
        Go over N/k elements in the original array and put them in the bucket based on their values.
4: Partition buckets into k sets, each has M/k buckets.
5: In parallel, each thread performs the following:
        Sort M/k non-empty buckets.
6: Visit the array of buckets in order and put the sorted elements back to the original array.
```

Figure 2. Parallel Bucket Sort Algorithm

4. OpenMP

OpenMP is a shared-memory application programming interface (API) whose features are based on prior efforts to facilitates shared-memory parallel programming [1]. The first version of OpenMP, consisting of a set of directives that could be used with Fortran, was introduced to the public in late 1997. OpenMP compilers began to appear shortly thereafter. Since that time, bindings for C and C++ have been introduced, and the set of features has been extended. Compilers are now available for virtually all SMP platforms. Although it is not an official standard, it is an agreement reached between the members of the ARB, who share an interest in a portable, user-friendly, and efficient approach to shared-memory parallel programming. ARB is a group of vendors who joined forces during the latter half of the 1990s to provide a common means for programming a broad range of SMP architectures. The number of vendors involved in maintaining and further developing its features has grown. Today, almost all the major computer manufacturers, major compiler companies, several government laboratories, and groups of researchers belong to the ARB. One of the biggest advantages of OpenMP is that the ARB continues to work to ensure that it remains relevant as computer technology evolves. OpenMP is under cautious, but active, development; and features continue to be proposed for inclusion into the application programming interface. Applications live vastly longer than computer architectures and hardware technologies; and, in general, application developers are careful to use programming languages that they believe will be supported for many years to come. The same is true for parallel programming interfaces.

OpenMP is intended for implementation not only on shared-memory multicore-, multithreading-processors machines but also on uniprocessor computers. OpenMP is not a new programming language because it is notation that can be added to a sequential program in Fortran, C, or C++ to describe how the work is to be shared among threads that will execute on different processors or cores and to order accesses to shared data as needed. The appropriate insertion of OpenMP features into a sequential program will allow many, perhaps most, applications to benefit from shared-memory parallel architectures, often with minimal modification to the code. In practice, many applications have considerable parallelism that can be exploited. The success of OpenMP can be attributed to a number of factors. One is its strong emphasis on structured parallel programming. Another is that OpenMP is comparatively simple to use because the burden of working out the details of the parallel program is up to the compiler. It has the major advantage of being widely adopted, so that an OpenMP application will run on many different platforms. Being timely is the strongest benefit that OpenMP offers. With the strong growth in deployment of both small and large SMPs and other multithreading hardware, the need for a shared-memory programming standard that is easy to learn and apply is accepted throughout the industry. The vendors behind OpenMP collectively deliver a large fraction of the SMPs in use today. Their involvement with OpenMP ensures its continued applicability to their architectures.

The parallel bucket algorithm is implemented in C using OpenMP runtime library.

5. EXPERIMENT RESULTS

5.1 CPU Runtime Measurement

Standard C function clock() was used to measure the CPU runtime required to run the test datasets. The function returns number of clock ticks consumed by the CPU. Clock ticks are units of time of a constant but system-specific length, as those returned by function clock. The following is an example of how to measure the time in seconds that the sorting algorithm consumes:

```
t0 = clock()
perform sorting
t1 = clock()
elapsed CPU time = (t1-t0) / CLOCK_PER_SEC
```

CLOCK_PER_SEC represents the number of clock ticks per second. Dividing a count of clock ticks by this expression yields the number of seconds the CPU has spent on sorting.

5.2 Results

The experiments were performed on a Macbook Pro machine, with 2.26 Ghz Intel Core 2 Duo processor and 8 GB of DDR3 memory. The test datasets are stored in text files and each consists of ten, twenty, thirty, forty, fifty, sixty, seventy, eighty, ninety, one hundred and two hundred million of unsorted positive integers. These integers, ranging from 0 to the maximum value of its dataset, were generated randomly using a different program from the main sorting programs. For example, ten million dataset would contain random values from 0 to 10,000,000. The test datasets are sorted one by one using sequential bucket sorting algorithm with one thread, and then parallel version with two and four threads. The results are shown as number of seconds in CPU time that bucket sorting algorithm took to perform on different datasets. Minor improvement was noticed when running the parallel algorithm using two and four threads compared to a single thread in the sequential version. The results are shown in

The results show some improvement when the test datasets are large enough to benefit from parallel computation. This is when the communication, synchronization overhead among the parallel threads is overcome by the speed up. Otherwise, sequential computation by single thread is preferred when the test datasets are small.

Since the testing was done on the system with only 2 physical processor cores, setting and running the algorithm with 4 threads does not help the performance. As a result, creating too many threads, regardless of the number of cores, can seriously degrade system performance because of overheads from starting and terminating threads and the race condition for occupying limited embedded hardware resource. Limitation of cache memory and data sharing architecture induce frequent data movement overhead between cache and main memory.

Table 1.

The results show some improvement when the test datasets are large enough to benefit from parallel computation. This is when the communication, synchronization overhead among the parallel threads is overcome by the speed up. Otherwise, sequential computation by single thread is preferred when the test datasets are small.

Since the testing was done on the system with only 2 physical processor cores, setting and running the algorithm with 4 threads does not help the performance. As a result, creating too many threads, regardless of the number of cores, can seriously degrade system performance because of overheads from starting and terminating threads and the race condition for occupying limited embedded hardware resource. Limitation of cache memory and data sharing architecture induce frequent data movement overhead between cache and main memory.

Table 1. CPU Runtime Comparison

Dataset (million)	1 Thread	2 Threads	4 Threads
10	0.6676	0.6586	0.6559
20	1.3757	1.3453	1.3417
30	2.1935	2.0395	2.0575
40	2.8608	2.7892	2.6905
50	3.5977	3.5348	3.5443
60	4.5591	4.4312	4.3836
70	5.3482	5.2341	5.3219
80	6.2249	6.0819	6.0926
90	7.4402	7.1979	7.0915
100	8.1446	7.9903	8.0638
200	20.046	19.1282	19.449

Figure 3 shows the chart of CPU runtime comparison when performing bucket sorting by one thread, two threads and four threads.

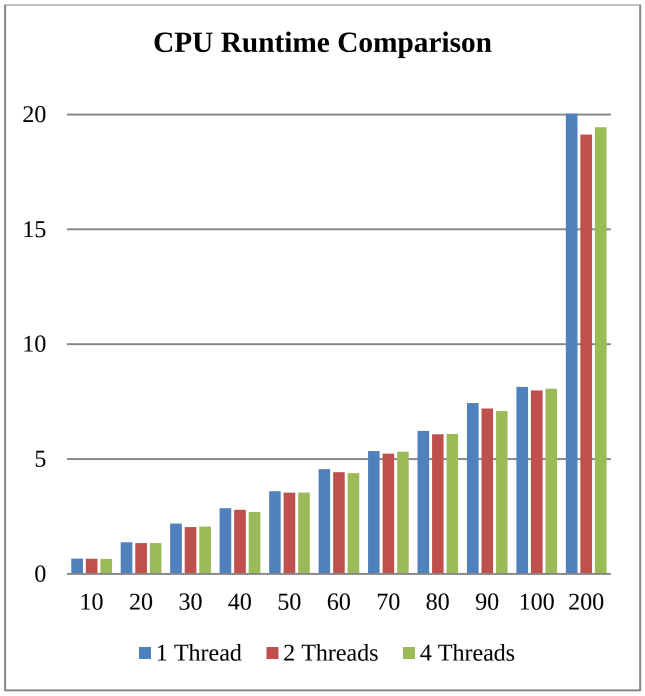

Figure 3. CPU Runtime Comparison

The speedup in the case of 200 million dataset is about 1.05, which is not much. This can be explained by the fact that only a small portion of the algorithm is parallelized. The rest of them still executes in sequential manner because their structures cannot be broken into independent parallel blocks. Once again, Amdahl's Law has proven its correctness in predicting the maximum speedup using multiple processors. Figure 4 shows Amdahl's Law speedup chart. In this experiment, less than 50% of the code is parallel. With only 2 processor cores, the results consistently match with the 50% curve in the speedup chart.

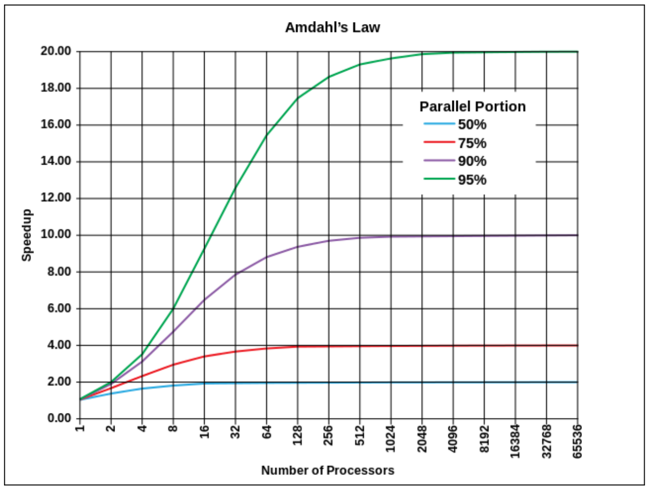

Figure 4. Amdahl's Law Speedup Chart

(Source: wikipedia.org)

6. CONCLUSION AND FUTURE WORK

Utilizing the evolving multicore processor architecture, this paper shows how the sequential bucket sorting algorithm can be converted to a parallel bucket sorting algorithm and be implemented in C and executed using OpenMP API. The experiment results show some improvement when the test datasets are large enough to benefit from parallel computation. This is when the communication and synchronization overheads among the parallel threads are overcome by the speedup. Otherwise, sequential computation by single thread is preferred when the test datasets are small. Since only a small portion of the algorithm is parallelized, the speedup gain is not significant, only about 1.05. Once again, Amdahl's Law has proven its correctness in predicting the maximum speedup using multiple processors. With less than 50% of the code in the implementation are parallelized, future work may be done by finding ways to crease the percentage of parallel code to benefit from the speedup suggested by the Amdahl's Law.

7. REFERENCES

[1] Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. Vol. 10. MIT press, 2008.

[2] Kim, Kil Jae, Seong Jin Cho, and Jae-Wook Jeon. "Parallel quick sort algorithms analysis using OpenMP 3.0 in embedded system." *Control, Automation and Systems (ICCAS), 2011 11th International Conference on*. IEEE, 2011.

[3] Mitzenmacher, Michael, and Eli Upfal. Probability and computing: Randomized algorithms and probabilistic analysis. Cambridge University Press, 2005.

[4] http://openmp.org/wp/

[5] Zhang, Keliang, and Baifeng Wu. "A novel parallel approach of radix sort with bucket partition preprocess." *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 2012.

[6] Zhang, Keliang, and Baifeng Wu. "Task scheduling for GPU heterogeneous cluster." *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*. IEEE, 2012.

[7] Zhang, Keliang, and Baifeng Wu. "Task Scheduling Greedy Heuristics for GPU Heterogeneous Cluster Involving the Weights of the Processor." *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE Computer Society, 2013.

[8] Zhao, Chengyan Chuck. "Explore Source-level Software-only Thread-Level Speculation Parallelism."

[9] http://en.wikipedia.org/wiki/Amdahl_s_law

[10] Zhao, Zhongxiao, and Chen Min. "An Innovative Bucket Sorting Algorithm Based on Probability Distribution." *Computer Science and Information Engineering, 2009 WRI World Congress on*. Vol. 7. IEEE, 2009.