Parallel Sorting Algorithms: A Survey

Manish Yadav (UFID: 3836-6483)

Abstract

In computer science, sorting algorithms have been a fundamental algorithm. A sorting algorithm puts the element of a list in a specified manner. Over the years, numerous sorting algorithms have been made to make the sorting algorithm efficient. In the past decade, there have been tremendous advances in the microprocessor. Going all the way from a few MHz to order of 8.7 GHz(AMD FX-8370). Not only CPUs, but GPUs have also been updated by the order of magnitude and are capable of carrying up to 13,448 GFLOPS. The role of concurrency has been realized and parallel processing is a cost-effective method to increase the efficiency of CPU and data-intensive problems. In this survey paper we will compare various sorting algorithms that could be run on multiple cores using CPUs or GPUs and how do they compare against a single core sorting algorithm.

1 Introduction

Sorting has been one of the ubiquitous problems in computer science. Because working with sorted data is far easier than with random data. Sorting has been defined as: A permutation (reordering) $< a_1^{'}, a_2^{'}...a_n^{'} >$ of the input sequence such that $< a_1^{'} \leq a_2^{'}... \leq a_n^{'} >$[CLRS09]. Sorting algorithms can be broadly categorized as a comparison based and non-comparison based. Lower bound of comparison-based sorting algorithms is $O(nlg(n))$ where n is the number of elements in the input.
Paralleling a sequential algorithm involves storing the distributed algorithm onto an available process so that it is being shared with other available processes. In a sequential algorithm, the data resides with the process's memory that makes it easier to access data. However, in the case of parallel processing, the data could reside either with a single process or shared among processes. In the sequential algorithm, the data resides within the main memory and hence can perform compare or exchange. In the case of parallel processing, the data could reside with a single process or shared among processes. If the data resides on the single process, the process is easier and as simple as a sequential processing algorithm. However, if the data resides on different processes, it is much more difficult to access it.

2 Algorithms

2.1 Comparison Based Sorting

2.1.1 Bitonic Sort

One of the classic parallel comparison based sort algorithm. It was developed by Kent E. Batcher in the year 1968[Bat68]. A *bitonic sequence* is defined as a list of ascending and descending numbers. For e.g 12 23 43 56 87 34 14 8 6 is considered to be a bitonic sequence. The main operation in bitonic sort works by merging two *bitonic sequence*.
There are two steps in bitonic sorting:
· *Creating Bitonic Sequence*: We divide a sequence into smaller chunks while maintaining the bitonic property of the sequence i.e the first sequence is less than to that in the second one. A bitonic sequence is divided into two halves with increasing order and decreasing order. An element a_i is compared with the n element, if they are out of order, they would be swapped.
· *Created sorted sequence from the bitonic sequence*: A pair of a consecutive element is considered to be a bitonic sequence. We start with a sequence of number which is partitioned in smaller sequence and then merged. Two adjacent bitonic sequences are sorted in ascending order, the next two are in descending order and so on. The process is continued until the entire array is converted into one single bitonic sequence

Analysis: Run time analysis of Bitonic sort is $O(lg^2(n))$ on a parallel processor. It is not data-dependent and hence it's a suitable candidate for sorting network in hardware or parallel processor array[Bat68]

2.1.2 Sample Sort

Another sorting algorithm that could be used over $nlg(n)$ processor is **bucket sort**. In this uniformly distributed over an interval, the array is divided into buckets of roughly n/p size, where p is the number of processors. The run time of this algorithm is $O(lg(n)$.
Unfortunately, real-world data is not evenly spaced over an interval and there may result in bucket overflow or underflow, thereby degrading performance. The sample sort works on the same principle as a quick sort. Just like quick sort selects the pivot element around which to partition the array [CLRS09], Sample sort works similarly, instead of choosing a single element as pivot, it takes a sample and partitions them into subarray around it.[FM70]

Algorithm 1: SAMPLESORT
Select a sample size **s**, sort the partition s and select $m-1$ elements from it that defines m **splitters**; Loop over the data and send each element to appropriate processor; Perform Sorting over these buckets;

Analysis:
$T_{samplesort}(N,P) = T_{subsample}(N,P) + T_{split}(N,P) + T_{locsort}(WS \cdot N/P)$
$T_{split}(N,P) = 2A \cdot P + 2A \cdot N/PlogP + R \cdot N/P$[FM70]

2.2 Non-Comparative Sorting Algorithm

In contrast with the above algorithm, we know will have a look at *non comparison-based sorting algorithms*

2.2.1 Radix Sort

Radix sort does not rely on comparisons to generate relative ordering of keys. Instead, it uses a representation of keys as b-bit integers. The basic radix sort implementation examines each digit at a time and uses a stable sort algorithm on each digit. The number of sorts starts with working on the most significant bit and then towards the least significant bit[CLRS09]. In case of parallel implementation of radix sort, instead of working on the number in digit by digit fashion, we work on r bits at a time, where $r < b$. During the i_{th} iteration, it sorts the i_{th} least significant block of r-bits. For radix sort to be stable, one must use a stable sorting algorithm to sort the r-bits.
Due to its nature of buckets, it one of the easiest and parallelizable algorithms to implement in practise[Ble90]. The parallel version of the counting sort[CLRS09] could be modified into radix sort. A $prefix - sum$ is sum of partial sum on a given sequence$(x[0]; x[0] + x[1]; x[0] + x[1] + x[2]; .....)$. When $prefix - sum()$ is applied on the array of bits of the number, i.e containing only 1s or 0s, it will return the number of 1s in the given array. A second $prefix - sum()$ is applied on the bits inverted which essentially gives us the total number of 0s in the given bit array $(diminished - prefix - sum())$. $diminished - prefix - sum()$ provides position new position for the number. The result of $prefix - sum()$ plus $diminished - prefix - sum()$ gives the final position of the the number in the sorted array.

Analysis: Computation of $prefix - sum$ and $diminished - prefix - sum()$ takes $(O(lg(n)))$ time with $n - 1$ processors and for a constant value of b and r

2.2.2 Enumeration Sort

Like Radix Sort, Enumeration Sort doesn't rely on the compare-exchange method. The core idea behind Enumeration Sort is to identify the rank of each element. The rank of an element a is defined as the number of elements smaller than a in sorted order. It takes n^2 processor by configured in a square array consisting of n rows consisting n processors in each row. The processors in row or column are interconnected to form a binary tree. Each processor at (row, column) can store elements of a sequence in the register at (row, column) and compare them. Using the binary tree construction of processors, it receives and sends data from another processor[Akl14].

Analysis: Due to binary tree order, multiple processors can compare values at once and the order of traversal in the tree is no more than $lg(n)$. Hence the overall complexity of Enumeration sort is $O(lg(n))$

3 Conclusion

While the single-core processor has a limitation of heat vs speed trade-off, multi-core processors have an advantage over the single-core processor. As the processing speed cannot be exponentially increased as this would lead to a large amount of heat dissipation over a very small area. To overcome this limitation, CPU/GPU manufacturers have added multiple cores to take help to parallelism to increase the processing power of processors. With the sequential algorithm, it is hard to beat the lower bound $O(nlg(n))$ for sorting in a comparison based model. However, with efficient work distribution among multiple cores, one could overcome the limitation of $(O(nlg(n)))$ and can achieve a performance of $(lg(n))$. I predict the number of cores on machines with continue to rise and to achieve a faster sorting algorithm. Having reviewed literature, I believe that moving from CPU to GPU based processor would have a significant impact soon. GPU based processor is not only helpful for graphic processing but is also useful for faster processing of floating-point operations. Multicore CPUs could be used to send data to a single GPU that processes a chunk of data much faster than a CPU.

References

[Akl14] Selim G Akl. *Parallel sorting algorithms*, volume 12. Academic press, 2014.

[Bat68] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.

[Ble90] Guy E Blelloch. *Vector models for data-parallel computing*, volume 356. MIT press Cambridge, 1990.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[FM70] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970.