View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Portal Jurnal Elektronik Universitas Kristen Satya Wacana (UKSW)

IJITEB
Vol. 1
No. 1
2018

International Journal of Information Technology and Business

http://ejournal.uksw.edu/ijiteb

Parallel Counting Sort: A Modified of Counting Sort Algorithm

Pratyaksa Ocsa Nugraha Saian[a*]

[a] Fakultas Teknologi Informasi, Universitas Kristen Satya Wacana

Keywords :
Counting Sort,
Parallel,
A Modified Algorithm

Abstract : Sorting is one of a classic problem in computer engineer. One well-known sorting algorithm is a Counting Sort algorithm. Counting Sort had one problem, it can't sort a positive and negative number in the same input list. Then, Modified Counting Sort created to solve that's problem. The algorithm will split the numbers before the sorting process begin. This paper will tell another modification of this algorithm. The algorithm called Parallel Counting Sort. Parallel Counting Sort able to increase the execution time about 70% from Modified Counting Sort, especially in a big dataset (around 1000 and 10.000 numbers).

1. Introduction

Sorting is one of many classic problems in a computer engineering. Although sorting usually being used in computer engineer, but sorting used in another field of study too. For example, sorting can be implemented in education [1][2], in biology [3], or even in economy [4][5] field of study. In computer engineering itself, sorting process can be used in many ways, such as network engineering [6][7], big data process [8][9], or in database process [10].

An algorithm is an object which created with a purpose to solve any problem in given circumstances [11]. Basically, a sorting algorithm is an algorithm to do the sorting process. Sorting algorithms had been created by many researchers. As for now, there are many algorithms already presented by them, such as Bubble Sort [12], Quick Sort, Merge Sort [13] and many more. Like two sides of a coin, that algorithms always have an advantage and a disadvantage for each one of them.

While many algorithms already presented, deciding which algorithm to be used is not that easy [14]. There are many consequences when choosing the wrong one. It can affect memory usage and increase the execution time of the application. Not only that, choosing which hardware to do the sorting process is something crucial too. Research [15] tells there is a significant difference between using high-end Central Processing Unit (CPU) and Graphic Processing Unit (GPU). GPU able to run 20 times faster than high-end CPU, but usually GPU is more expensive than high-end CPU.

In 2009, Cormen et al. present a new sorting algorithm in their book. The algorithm didn't use comparing method to get the sorted list. Instead, the algorithm will count the appearance of the value in the list. Therefore, the algorithm called Counting Sort algorithm [16]. Like any sorting algorithm, Counting Sort will have a list (usually an array) of integer number or character and the algorithm will try to arrange it in any given order (ascending or descending). Counting Sort assumes every element in the array contain a number from zero to n where n is a positive integer number. It makes counting sort algorithm can't sort both negative and positive number in one array. In another research [17], it can be solved by dividing the negative, zero, and positive number into different arrays and then the arrays will be sorted one by one. Then, these arrays will merge into one big array which will have a sorted number. Later in this paper, this algorithm called Modified Counting Sort algorithm.

This paper tells another modification of the Modified Counting Sort algorithm. The main idea of the algorithm coming from Idrizi et al.'s algorithm [17]. Based on that algorithm, this experiment trying to enhance and optimize it more. Instead of sorting the array one-by-one, this algorithm will sort all of them simultaneously using more than one threads. This algorithm should reduce the execution time of

* Corresponding authors
e-mail addresses : pratyaksa.ocsa@uksw.edu

the sorting process because there are at least two process runs at the same time.

The rest of the paper is structured as follows: Section 2 tells about several theories and previous work which related with this paper, Section 3 tells about the modified algorithm, and Section 4 tells about the conclusion and future works related to this paper.

2. The Material and Method

Before discussing more the modified algorithm, there are some theory or material which important and need to be discussed. All of them are have high relevance with this paper.

Research conducted by Muhammad Ezar Al Rivan tells about how good a combination of several sorting algorithms is. This research use five different algorithms, which is Quick Sort, Merge Sort, Insertion Sort, Bubble Sort, and Selection Sort. From five algorithms, the researchers divided them into two group of sorting algorithm. Quick Sort and Merge Sort is in group A and Insertion Sort, Bubble Sort, Selection Sort is in group B. Researcher choose one algorithm from group A and combined it with one algorithm from Group B. To check how well that combination, researcher will measure the execution time of it. After doing it to every possibility of combination, it appears the combination of Merge-Insertion Sort and Merge-Selection Sort have the best execution time of all [18].

Another research conducted by Dwi M J Purnomo, et al. tells about an implementing a Bubble Sort in Field Programmable Gate Array (FPGA). The Bubble Sort itself implemented in both serial and parallel programming. They measure the memory usage and execution time. It appears that serial Bubble Sort have better memory usage than parallel Bubble Sort, but parallel Bubble Sort have better execution time than serial Bubble Sort [19].

Another research conducted by Ivan Kamarov, et al. tells about implementation of brute force algorithm to create k-Nearest Neighbor Graph (k-NNG). Then, this algorithm implemented into Graphics Processing Unit (GPU) combined with a quicksort algorithm. The result of this experiment is a combination of a brute force algorithm and quicksort algorithm in GPU able to process larger data in a better time execution [20].

To give a better understanding of what is this paper about, it is important to know about some theory. This paper will explain more about the sorting algorithm, counting sort algorithm, and parallel counting sort algorithm.

2.1. Sorting Algorithm

In computer science, a sorting algorithm is an algorithm to rearrange some list in a specific order. The list can be an array, a vector, or any data type that can be stored more than one element at the same place. Integer or Char data type is commonly used in any sorting process. Sorting algorithm will always produce an arranged list in ascending or descending order. Ascending list is a list which its values are come from "a small number" to "a big number" while descending list is a list which its values come from "a big number" to "a small number".

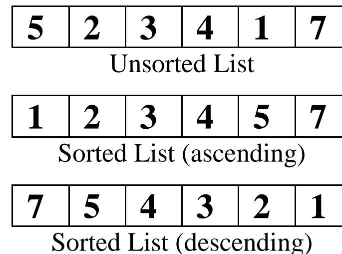

Fig. 1 Example of an unsorted list, sorted list (ascending and descending)

Fig. 1 is an example of an input and an output of the sorting algorithm. The unsorted list contains several elements of number (ex: 5, 2, 3, 4, 1, 7) and that list has values in random order – not in ascending or descending order. In some cases, that list needs to be arranged properly to get a better information. That is how any sorting algorithm works. That list will be arranged by any sorting algorithm, then the result will always in a good order. It can be an ascending order (1, 2, 3, 4, 5, 7) or descending order (7, 5, 4, 3, 2, 1).

Researcher tends to measure how good any sorting algorithm is. They usually consider it from several things, such as running time/execution time or how much memory needed to do the sort process. In this paper, only the running time/execution time is chosen to be a benchmark for any algorithm to be tested.

2.2. Counting Sort Algorithm

Counting Sort algorithm always starts with one list of an unordered integer numbers (List A). Then, it will create another list to save of how many times the number appears in the List A (List B). After both of lists successfully created, the algorithm will do the counting process. It will go through in each element of List A to count the appearance number and save it in List B. Now, every element in List B contains a number and that number is the "correct position" of the number in List A. Finally, the algorithm will create one last list (List C) to save the "correct position" of the number in List A. The algorithm will match each of numbers in List A with its position in List B and save it in List C. Implementation of the Counting Sort algorithm can be seen in Fig. 2.

```
void CountingSort() {
   int[] input = InitializeArrayToBeSorted(); /* Generate a random number */
   int[] count = int[input.length];  /*To be used in counting process*/
   int[] output = int[input.length]; /*To be used as a sorted list */
   /*Count the appearance of the number*/
   for(int i=0;i<input.length;i++) { ++count[input[i]]; }
   /*Rearrange the list into a sorted list*/
   for(int i=0;i<input.length;i++) {
      output[count[input[i]]-1] = input[i];
      --count[input[i]];
   }
}
```

Fig. 2 Pseudocode of Counting Sort Algorithm

To measure how good this counting sort algorithm, like any algorithm it will be used the time complexity of the algorithm. The time complexity of a Counting Sort algorithm is $O(n + k)$ [16] where n is the number of elements in an array and k is the range of the input. The range of the input is the range between the smallest number and the biggest number in List A.

2.3. Parallel Counting Sort Algorithm

As explained in Section 1 before, the problem of counting sort appears when there are a negative integer value appears in the List A of Counting Sort. This problem can be solved by split the list into a negative list and a positive list. The flowchart of this process can be seen in Fig. 3.

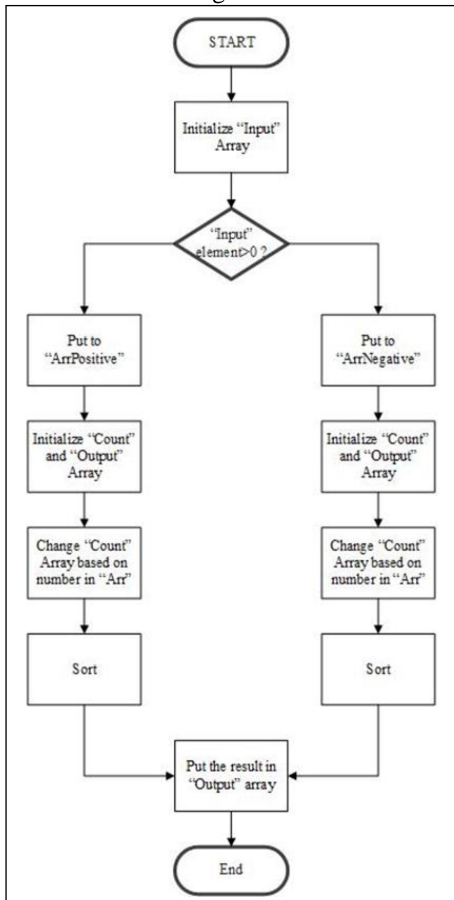

Fig. 3 Flowchart of Modified Counting Sort Algorithm

From flowchart in **Fig. 3** it tells that the splitting process to distinguish between a positive number and a negative number happen before the sorting process. Every element in unordered list will be checked if the number is greater than zero or not. If the number is greater than zero, then it will be stored in the "ArrPositive" list and if the number is smaller than zero, then it will be stored in the "ArrNegative" list. Both of this will sort separately and the result of both will joined into one list again. The implementation of this process can be seen in **Fig. 4**.

```
void ModifiedCountingSort() {
   int[] input =
InitializeArrayToBeSorted();
   List ArrPositive;
   List ArrNegative;
   foreach (int i in input) {
      if(i < 0) {
        ArrPositive.add(i);
      } else {
        ArrNegative.add(i);
      }
   }
   //counting sort process
   …
}
```

Fig. 4 Pseudocode of Modified Counting Sort

This paper will tell another modification of this algorithm. In the Modified Counting Sort before, after the input list separated into "ArrPositive" and "ArrNegative", the sorting process run to both separately too. This sorting process runs in sequentially. Usually "ArrPositive" will be sorted first and "ArrNegative" next. Instead of works in two lists sequentially, this new algorithm will do the counting sort simultaneously. The detailed process of this algorithm can be seen in **Fig. 5**.

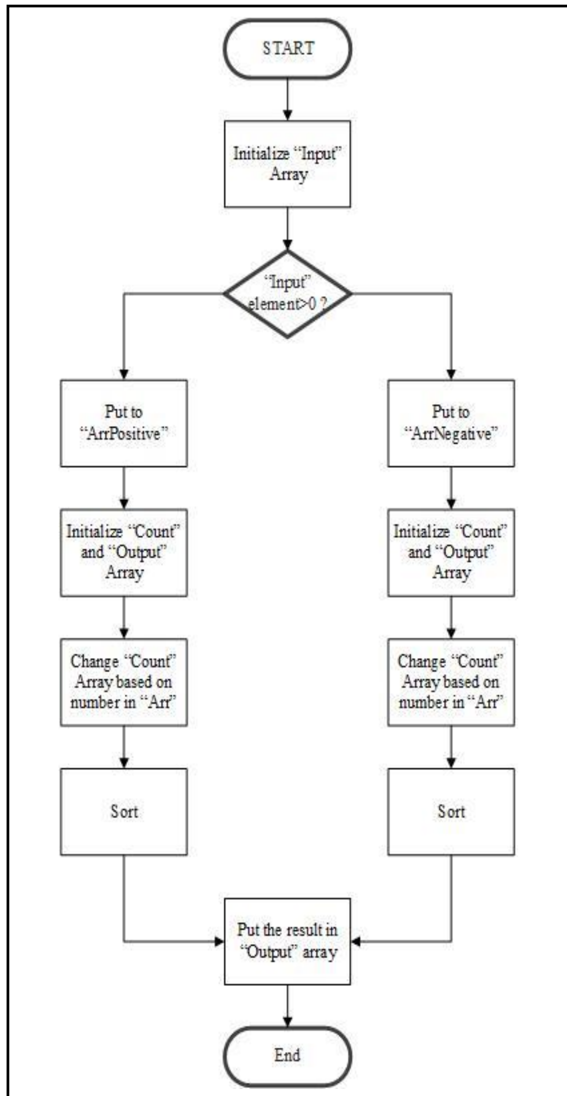

Fig. 5 Flowchart of Parallel Counting Sort Algorithm

Fig. 5 shows about the difference (marked with dotted line) from Modified Counting Sort Algorithm. Parallel Counting Sort will do the sorting process simultaneously. By doing it, Parallel Counting Sort should run faster than Modified Counting Sort Algorithm. The implementation of Parallel Counting Sort algorithm can be seen in **Fig. 6**.

```
void ParallelCountingSort() {
   int[] input =
InitializeArrayToBeSorted();
   //split the input array
   …
   Thread t[] = CreateThread();
   t[0] =
DoModifiedCountingSort(ArrPositive);
   t[1] =
DoModifiedCountingSort(ArrNegative);
   …
   output = ResultOf(t[0]) + ResultOf(t[1]);
   …
}
```

Fig. 6 Pseudocode of Parallel Counting Sort

The main difference between Modified Counting Sort and Parallel Counting Sort is in the thread creation. This thread has never been created in Modified Counting Sort, but in Parallel Counting Sort, it will create two new threads. These two threads used to enable the computer to do any process simultaneously. One thread will handle the sorting process for "ArrPositive" and another thread will handle "ArrNegative". The algorithm will wait until both threads finished do the sorting process, then the result will be merged into one list.

3. Result and Discussion

This section mainly talks about the testing process and how to compare the result of how well both algorithms to solve a sorting problem in many test cases. The Modified Counting Sort Algorithm and Parallel Counting Sort will be tested in a similar condition. Testing process held in a computer with hardware specification: Intel Core i5-3210M CPU @2.50GHz and 4GB RAM. The computer uses an operating system: Windows 10 Education 64-bit (10.0, Build 17134).

The testing process proceeds in three steps: (1) preparing test cases, (2) running the algorithm with the prepared test cases, and (3) getting the execution time.

3.1. Preparing Test Cases

There are some test cases prepared to measure the execution time of each algorithm. Both will get different input numbers, from ten, a hundred, a thousand, a ten thousand, and a hundred thousand of integer numbers. It contains positive numbers, more than one zero, and negative numbers which all of them will be generated randomly.

```
int numbers[NUMBERS];
//Generate Random Number
std::random_device rd;  //Will be used to
obtain a seed for the random number engine
std::mt19937 gen(rd()); //Standard
mersenne_twister_engine seeded with rd()
std::uniform_int_distribution<>
dis(-(MAX_RANGE - 1), MAX_RANGE-1);
for (int n = 0; n < NUMBERS; ++n) {
        numbers[n] = dis(gen);
}
```

Fig. 7 C++ source code to generate random number

Error! Reference source not found. tells about a source code of the implementation of generating random numbers. It started with preparing an array (or a list) to be used as input numbers later. Then, by using C++ standard library function (all of them included in "random" header), the numbers generated one by one until all places in input numbers filled. Then, this function needs a little modification to gain control of "how random the generated number". The modification is by putting a control variable (*MAX_RANGE*) so the random

number will always be in the desired range, which is –(*MAX_RANGE*-1) to (*MAX_RANGE*-1).

3.2. Running the Algorithm

Each set of randomly generated number from the previous step will be used for each algorithm as an input. To maintain the validity, both algorithm will use the same set of randomly generated number.

3.3. Getting the Execution Time

The last step of the testing process is getting the execution time of both algorithms. It will be used to measure the difference between them and decide which algorithm have a better execution time. In this experiment, the C++ programming language is used to get the execution time. The implementation of how to get the execution time is shown in **Fig. 8.**

```
…
   clock_t tStart = clock();
   /* Parallel Counting Sort or Modified
Counting Sort algorithm */
   printf("Time taken: %.9fs\n",
(double)(clock() - tStart)/CLOCKS_PER_SEC);
   return 0;
…
```

Fig. 7 C++ source code to take execution time

Fig. 7 shows to get the execution time, in C++ use *clock()* function. This function is a "prepared" function and can be used by including "*time.h*" header. Then, the timer will be started when the algorithm about to started and finished when the algorithm finished too. The exact execution time obtained by finding the difference between start time and finish time. By doing this, the execution time will appear in milliseconds (ms).

3.4. Result

After doing the testing process in all test cases, the result of this experiment can be found in **Table 1.**

Table 1 Execution Time Result for Both Algorithm

Algorithm	Execution Time from Each Case				
	10	**100**	**1.000**	**10.000**	**100.000**
Modified Counting Sort	1ms	2ms	4ms	84ms	671ms
Parallel Counting Sort	1ms	2ms	3ms	18ms	161ms

Table 1 shows the execution time of both algorithms in each test cases. From test case 1: both algorithms get the same results, they need 1ms to sort 10 different numbers. From test case 2: Modified Counting Sort and Parallel Counting Sort run in 2ms. From test case 3: there is a slight difference between Modified Counting Sort and Parallel Counting Sort, the difference only 1ms. From test case 4: there is a significant gap between them. Parallel Counting Sort only needs 18ms while Modified Counting Sort needs 84ms to finish the sorting process. From test case 5: the gap gets wider; Modified Counting Sort need 671ms while Parallel Counting sort needs 161ms.

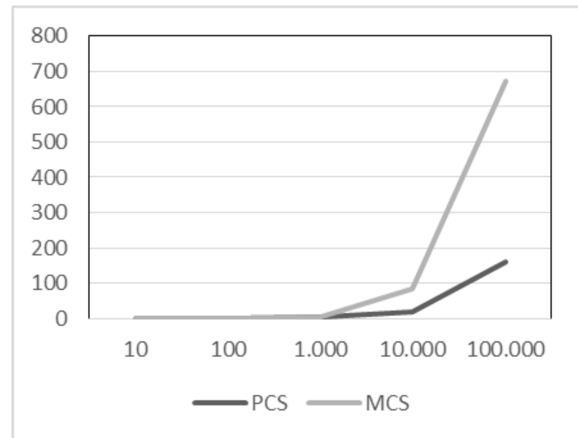

Fig. 8 Execution Time Chart

Fig. 8 showing the result of the experiment on both algorithm. The results show that in a relatively small set of randomly generated number (10, 100, and 1000 numbers) the result doesn't show a big difference. The gap distance starting to get wider after the algorithm get a big set of data (10.000 and 100.000 numbers) as an input. As described in the chart, more data being used, the gap gets wider too.

4. Conclusion

Based on the result explained before, Parallel Counting Sort able to have smaller execution time than Modified Counting Sort, especially in a big set of data. Parallel Counting Sort able to increase the execution time around 78.57% time in test case number 4 and around 76% in test case number 5. In a small set of data, the result tends to be the same since the execution time almost similar.

For the future works, Parallel Counting Sort needs to be compared with another sorting algorithm. To be more interesting, instead only comparing the execution time, the algorithm also comparing memory usages of each algorithm.

5. References

[1] L. Végh and V. Stoffová, “Algorithm animations for teaching and learning the main ideas of basic sortings,” Informatics Educ., vol. 16, no. 1, pp. 121–140, 2017.
[2] Z. KATAI, L. TOTH, and A. K. ADORJANI, “Multi-Sensory Informatics Education,” Informatics Educ., vol. 13, no. 2, pp. 225–240, Sep. 2014.
[3] G. Regalia, S. Coelli, E. Biffi, G. Ferrigno, and A. Pedrocchi, “A Framework for the Comparative Assessment of Neuronal Spike Sorting Algorithms towards More Accurate Off-Line and On-Line Microelectrode Arrays Data Analysis,” Comput. Intell. Neurosci., vol. 2016, pp. 1–19, 2016.
[4] X. Yang, Z. Zeng, R. Wang, and X. Sun, “Bi-objective flexible job-shop scheduling problem considering energy consumption under stochastic processing times,” PLoS One, vol. 11, no. 12, pp. 1–14, 2016.
[5] M. Kessel and C. Atkinson, “Ranking software components for reuse based on non-functional properties,” Inf. Syst. Front., vol. 18, no. 5, pp. 825–853, 2016.
[6] M. Codish, L. Cruz-Filipe, T. Ehlers, M. Müller, and P. Schneider-Kamp, “Sorting networks: To the end and back again,” J. Comput. Syst. Sci., vol. 1, pp. 1–18, 2016.
[7] F. Frattolillo, “A deterministic algorithm for the deployment of wireless sensor networks,” Int. J. Commun. Networks Inf. Secur., vol. 8, no. 1, pp. 1–10, 2016.
[8] H. Mohammed, N. Clarke, and F. Li, “An Automated Approach for Digital Forensic Analysis of Heterogeneous Big Data,” J. Digit. Forensics, Secur. Law, 2016.
[9] D. H. S. Chung, P. A. Legg, M. L. Parry, R. Bown, I. W. Griffiths, R. S. Laramee, and M. Chen, “Glyph sorting: Interactive visualization for multi-dimensional data,” Inf. Vis., vol. 14, no. 1, pp. 76–90, 2013.
[10] Y. S. Chang, R. K. Sheu, S. M. Yuan, and J. J. Hsu, “Scaling database performance on GPUs,” Inf. Syst. Front., vol. 14, no. 4, pp. 909–924, 2012.
[11] R. K. Hill, “What an Algorithm Is,” Philos. Technol., vol. 29, no. 1, pp. 35–59, 2016.
[12] Reina and J. B. Gautama, “Perbandingan Bubble Sort dengan Insertion Sort pada Bahasa Pemrograman C dan Fortran,” ComTech, vol. 4, no. 2, pp. 1106–1115, 2013.
[13] Arief Hendra Saptadi and D. W. Sari, “Analisis algoritma insertion sort, merge sort dan implementasinya dalam bahasa pemrograman c++,” J. Infotel, vol. 4, no. 2, pp. 1–8, 2012.
[14] A. Taherkhani, A. Korhonen, and L. Malmi, “Categorizing variations of student-implemented sorting algorithms,” Comput. Sci. Educ., vol. 22, no. 2, pp. 109–138, Jun. 2012.
[15] E. Avramidis and O. E. Akman, “Optimisation of an exemplar oculomotor model using multi-objective genetic algorithms executed on a GPU-CPU combination,” BMC Syst. Biol., vol. 11, no. 1, pp. 1–24, 2017.
[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., no. 2. London: The MIT Press Cambridge, Massachusetts, 2009.
[17] F. Idrizi, A. Rustemi, and F. Dalipi, “A new modified sorting algorithm: A comparison with state of the art,” in 2017 6th Mediterranean Conference on Embedded Computing, MECO 2017 - Including ECYPS 2017, Proceedings, 2017, no. June, pp. 1–6.
[18] M. Ezar and A. Rivan, “Perbandingan Kecepatan Gabungan Algoritma Utama Quick Sort dan Merge Sort dengan Algoritma Tambahan Insertion Sort , Bubble Sort dan Selection Sort,” J. Tek. Inform. dan Sist. Inf., vol. 3, no. 2, pp. 319–331, 2017.
[19] D. M. J. Purnomo, A. Arinaldi, D. T. Priyantini, A. Wibisono, and A. Febrian, “Implementation of Serial and Parallel Bubble Sort on FPGA,” J. Ilmu Komput. dan Inf., vol. 9, no. 2, p. 113, Jun. 2016.
[20] I. Komarov, A. Dashti, and R. M. D’Souza, “Fast k-NNG construction with GPU-based quick multi-select,” PLoS One, vol. 9, no. 5, 2014.