World Academy of Science, Engineering and Technology
International Journal of Mathematical and Computational Sciences
Vol:5, No:3, 2011

Performance Comparison of Parallel Sorting Algorithms on the Cluster of Workstations

Lai Lai Win Kyi, Nay Min Tun

***Abstract*—**Sorting appears the most attention among all computational tasks over the past years because sorted data is at the heart of many computations. Sorting is of additional importance to parallel computing because of its close relation to the task of routing data among processes, which is an essential part of many parallel algorithms. Many parallel sorting algorithms have been investigated for a variety of parallel computer architectures. In this paper, three parallel sorting algorithms have been implemented and compared in terms of their overall execution time. The algorithms implemented are the odd-even transposition sort, parallel merge sort and parallel rank sort. Cluster of Workstations or Windows Compute Cluster has been used to compare the algorithms implemented. The C# programming language is used to develop the sorting algorithms. The MPI (Message Passing Interface) library has been selected to establish the communication and synchronization between processors. The time complexity for each parallel sorting algorithm will also be mentioned and analyzed.

***Keywords*—** Cluster of Workstations, parallel sorting algorithms, performance analysis, parallel computing, MPI.

I. Introduction

SORTING is one of the most common operations performed by a computer. Because sorted data are easier to manipulate than randomly-ordered data, many algorithms require sorted data. Moreover sorting is one of the most important operations in database systems and its efficiency can influences drastically the overall system performance. Sorting is of additional importance to parallel computing because of its close relation to the task of routing data among processes, which is an essential part of many parallel algorithms. To speed up the performance of database system, parallelism is applied to the execution of the data administration operations. The workstations connected via a local area network allow to speed up the application processing time [4].

Today, clusters of loosely coupled desktop computers present extremely popular infrastructure for development of parallel algorithms. The processes, running on computers in cluster, communicate with each other through messages. MPI is standardized and portable implementation of this concept, providing several abstractions that simplify the use of parallel computers with distributed memory.

Although, the majority of today clusters run on Linux operating systems, Microsoft Windows operating coupled with the .NET platform is also becoming an interesting alternative. The .NET platform, designed to simplify the connection of information, people, systems and devices has two important parts: (i) Common Language Infrastructure (CLI), a layer built upon operating system, allowing development of operating system independent applications and (ii) new programming language C# – simple, safe, object-oriented, network centered high performance language.

In this paper implementation of parallel sorting algorithms in aspect of linking Message Passing Interface to the C# and .NET framework is considered. The research has been carried out on performance evaluations of parallel sorting algorithms on the cluster of workstations. In next section MPI binding C# on .Net Platform and parallelization of sorting algorithms are presented. Furthermore, sorting algorithms detail with emphasis on technology is exposed in section three. In section four, the experimental setup and results in terms of computational times are given. The main findings are concluded in the last section.

II. Related Work

This section describes the related work to implement parallel sorting algorithms using MPI and C# on .NET platform.

A. Binding C# and MPI on .Net Platform

The language C# is object oriented language with bounds checking and garbage collection. Thus it helps writing safe code by protecting from dangerous pointer and memory-management errors, such as accessing the element of array out of its bounds or problems connected to creation and deletion of objects. The meta code, produced by C# compiler is then executed by the CLI interpreter, available for Windows systems and also for Linux systems [5].

Due to the fact that MPI standard only requires source compatibility and that current MPI implementations do not support .NET platform, the final code, using MPI libraries written in C language is not platform independent. On Windows systems the freely available MPICH library [2] is mostly used.

Besides the compatibility issues, there are also some problems regarding the binding of the MPI libraries to the C#

Lai Lai Win Kyi is with the Ph.D 7th Batch Candidate from Department of Information Technology, Mandalay Technological University, Mandalay Region, Myanmar. (e-mail: laelae83@gmail.com)

Nay Min Tun is with the Principle in Computer University, Kyaing Tong, Myanmar. (e-mail: naymin.300777@gmail).

language. First, the objects in .NET can be arbitrarily moved by garbage collector, and this must be prevented when they are in use by MPI functions. The solution, which still generates safe code, is to use special C# class to pin the object in some memory location and than obtain pointer to that object needed by MPI library functions. Special care is needed to unpin the object when it is not needed anymore. Secondly, MPI data types and constants are defined in C++ header files, which cannot be directly imported into C#. Therefore, MPI constants need to be represented as functions, which can return the value of particular constant on startup. Similarly special functions are written to create, access and delete special MPI data types. In order to put the binding problems out of the C# programmer's sight, a wrapper was written in C# and partially in C [6], providing an interface to the MPICH library that looks more like a normal C# class. It is reported that with careful pinning and unpinning of objects the performance of the MPI is only slightly affected [6].

B. Parallelization of Sorting Algorithms

There are six main steps in parallelizing sorting algorithms shown in Figure 1. Firstly, "Initialize MPI" is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI Environment. MPI.Init is called once typically at the start of the program before any other MPI functions are called. This provides application developers with the function declarations for all MPI functions. The second step is accepting data. If the node is the root of the cluster, it must open the data file and accept the data into the array of the same data type. Initially, the data file has already created by inserting large amount of data randomly at a root node. The root node also accepts the search key from user it must send it to all the other compute nodes.

To send the data of any type to all the other nodes, MPI.Bcast() function is used.For parallel processing to take place, the root of the cluster divides the data into multiple partitions according to the number of the compute nodes and distributes them to the compute nodes. To divide the entire data file into multiple fragments, the entire size of the file and the total number of nodes in the cluster must be known. The total number of nodes in the cluster must be determined by MPI.Comm.size(). The root node sends the data partitions to the compute nodes by calling the MPI.Scatter(). It can also use MPI.Send() and MPI.Recv() functions to send and receive data from one node to another.

After that the data file is partitioned according to the number of compute nodes at the root node. This data partition is sorted in parallelizing with any sorting algorithm by all computing nodes. Then the sorted data partitions are gathered by the root node using MPI.Gather() commands. MPI.Finalize allows the program to take MPI.Finalize is called at the end of the computation; it performs various clean-up tasks to terminate the MPI Environment. No MPI calls may be performed after MPI.Finalize has been called, not even MPI.Init

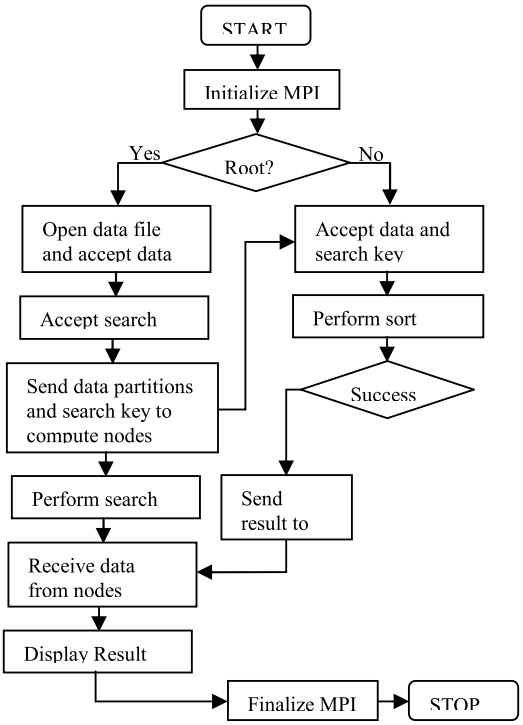

Fig. 1 Parallelization of Sorting Algorithms

III. PARALLEL SORTING ALGORITHMS

There are many types of sorting algorithms for parallel computing. In this paper, three parallel sorting algorithms will be implemented and evaluated. These algorithms are odd-even transposition sort, parallel merge sort and Parallel rank sort.

A. Odd-Even Transposition Sort

The Odd-even transposition sort algorithm [5,6] starts by distributing n/p sub-lists (p is the number of processors) to all the processors. Each processor then sequentially sorts its sub-list locally. The algorithm then operates by alternating between an odd and an even phase, hence the name odd-even. In the even phase, even numbered processors (processor i) communicate with the next odd numbered processors (processor i+1).

In this communication process, the two sub-lists for each two communicating processes are merged together. The upper half of the list is then kept in the higher number processor and the lower half is put in the lower number processor. Similarly, in the odd phase, odd number processors (processor i) communicate with the previous even number processors (i-1) in exactly the same fashion as in the even phase. It is clear that the whole list will be sorted in a maximum of p stages. Figure 2 shows an illustration of the odd-even transposition algorithm.

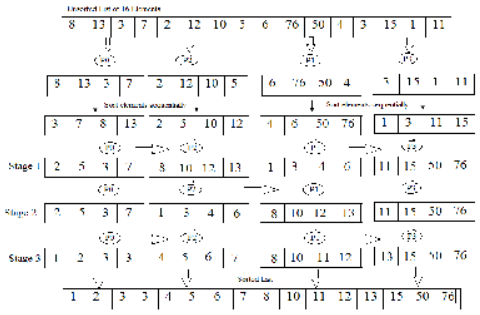

Fig. 2 Odd-Even Transposition Sort

Below is the analysis of the time complexity for the odd-even transposition sorting algorithm [3]. The performance of the sequential sort algorithm is:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(an^2)$$

where a=1/2.

The performance of the odd-even transposition algorithm is:

$$\sum_{i=1}^{n/p} i = 1 + 2 + 3 + \cdots + \frac{n}{p} = \frac{n/p(n/p-1)}{2} = \frac{n^2}{2p^2} - \frac{n}{2p} = O(bn^2)$$

where $b=1/2p^2$.

This means that theoretically speaking the time will reduce by 1/p2.

B. Parallel Merge Sort

Parallel merge sort algorithm uses a parallel binary merge sort strategy to sort its element. Parallel binary merge sort is composed of two phases: local sort and final merge. The local sort phase is carried out independently in each processor. Local sorting in each processor is performed as per normal serial external sorting mechanism.

After local sort phase is completed, the second phase: final merge phase starts. The merging phase is pipelined, instead of concentrating on one processor. The way the merging phase works is taking the results from two processors, and merging the two in one processor. As this merging technique uses only two processors, this merging is called “Binary Merging”. The result of the merging between two processors is passed on to the next level until one processor left; that is the host. Subsequently, the merging process forms a hierarchy. Figure 3 gives an illustration of the process[3].

The main motivation to use parallel binary-merge sort is that the merging workload is spread to a pipeline of processors, instead of one processor. It is true however that final merging has still to be done by one processor.

Sequential merge sort time complexity is $T_{serial} = (n-1)$ when parallelizing the merge sort algorithm the time complexity reduces to $T_{parallel}= 2^{n-p} –p+1$.

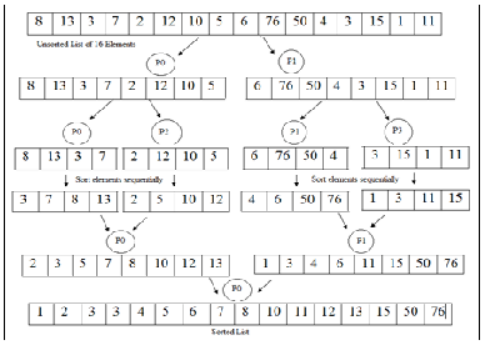

Fig. 3 Parallel binary-merge sort

C. Parallel Rank Sort

In the sequential rank sort algorithm (also known as enumeration sort), each element in the list to be sorted is compared against the rest of the elements to determine its rank amongst them [7]. This sequential algorithm can be easily parallelized by enabling the master processor to distribute the list amongst all the processors and assigning each slave processor n/p elements (where n is the list size and p is the number of processors). Each processor is responsible of computing the rank of all the n/p elements. The ranks are then returned from the slaves to the master who in turn is responsible of constructing the whole sorted list as shown in Figure 4.

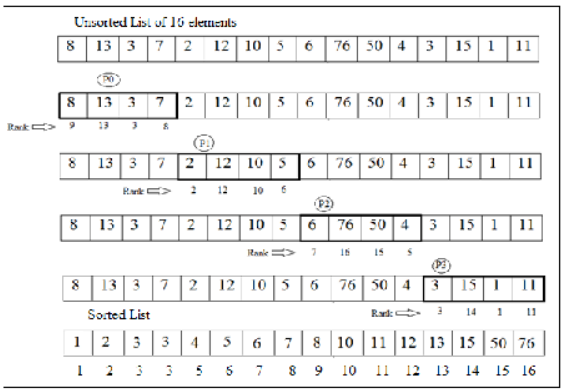

Fig. 4 Parallel Rank Sort Algorithm

In the sequential version of the rank sort algorithm, each element is compared to all the other elements. The complexity of the algorithm can be expressed as:

$$\sum_{i=1}^{n} n = O(n^2)$$

When parallelizing this algorithm, it can be easily seen that the complexity reduces to:

$$\sum_{i=1}^{n/p} n = O(cn^2), c = \frac{1}{p}$$

This means that if n number of processors is used then the

sorting time will become almost linear O (n).

IV. Results and Discussions

Each of the parallel algorithms stated above will be compared to its sequential implementation and evaluated in terms of its overall execution time, speedup and efficiency. The speedup is used to measure the gain of parallelizing an application versus running the application sequentially and can be expressed as:

$$\text{Speedup} = \frac{\textit{Execution time using one processor}}{\textit{Execution time using p processor}}$$

On the other hand, the efficiency is used to indicate how well the multiple processors are utilized in executing the application and can be expressed as:

$$\text{Efficiency} = \frac{\text{Execution time using p processor}}{\text{Total number of processor}}$$

The C# programming language and MPI library used to develop the sorting algorithms. The performance of the sorting algorithms was evaluated on a homogeneous cluster of workstations, with Window operating system. Each sorting algorithm performance was evaluated for 2, 4, 8, 16 and 32 processors. The speedup and efficiency will be calculated based on the previous records. An array of 2^{14} random integers was used to test the parallel algorithms.

A. Odd-Even Transposition Sort

The time of sequential odd-even transposition sort is 5×10^6 seconds. Figure 5 shows the total execution time for the odd-even transposition sorting algorithm. It can easily be seen that parallel algorithm is by far faster than the sequential sort algorithm. The speed up for the odd-even transposition sorting algorithm is also displayed in figure 6 along with the efficiency in figure 7.

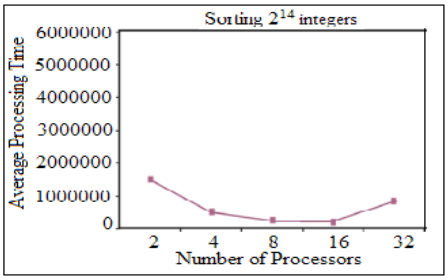

Fig. 5 Total Execution Time of the Odd-Even Transposition Sort Algorithm

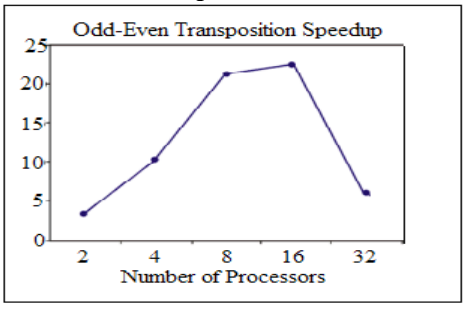

Fig. 6 Speedup of the Odd-Even Transposition Sort Algorithm

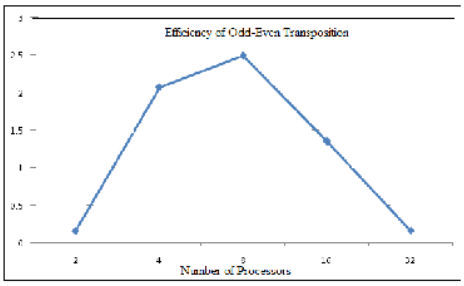

Fig. 7 Efficiency of the Odd-Even Transposition Sort Algorithm

B. Parallel Merge Sort

Parallel merge sort is one of the most efficient algorithms for sorting elements. The time of sequential merge sort is 29000 seconds. In Figure 8 an illustration of the total execution time of the parallel algorithm is displayed.

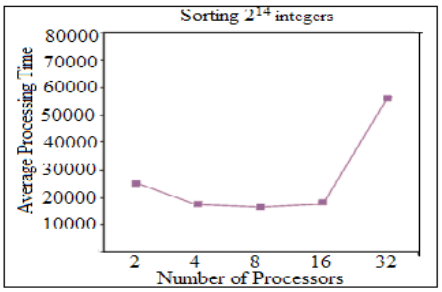

Fig. 8 Total Execution Time of the Parallel Merge Sort Algorithm

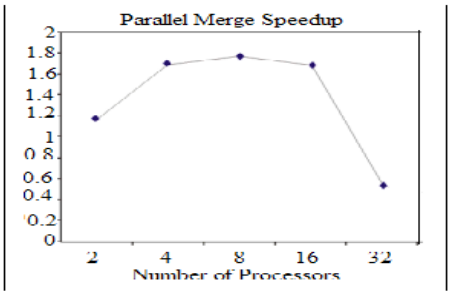

Fig. 9 Speedup of the Parallel Merge Sort Algorithm

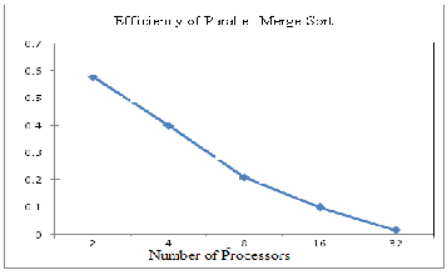

Fig. 10 Efficiency of the Parallel Merge Sort Algorithm

It shows that sorting using up to16 processors is helpful in reducing the total time required to sort the elements. However, increasing the processors to more than 16 processors will result in lower performance compared to the sequential merge sort algorithm. This of course is due to the

communication overhead that occurs between the processors to merge the result in to one sorted list. The speedup and efficiency of the parallel merge sort algorithm are displayed in Figure 9 and Figure 10 respectively.

C. Parallel Rank Sort

The time of sequential rank sort is 3.4×10^6 seconds. Running the parallel rank sort algorithm on 2 processors to sort 2^{14} integers is slower than the sequential implementation due to the communication overhead needed to distribute the whole unsorted list to all the processors. However, the benefit of parallelization kicks in after increasing the number of processors. Using 2 processors run parallel rank sort should increase the performance of the algorithm conditioned the number of elements to be sorted are greater than 2^{14} elements.

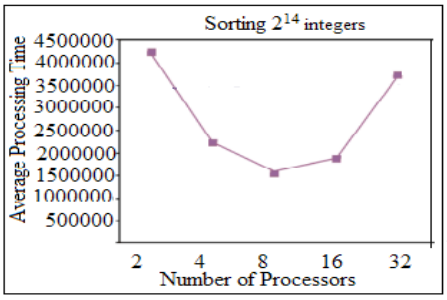

Fig. 11 Total Execution Time of the Parallel Rank Sort Algorithm

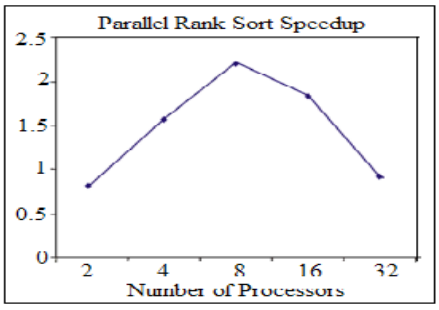

Fig. 12 Speedup of the Parallel Rank Sort Algorithm

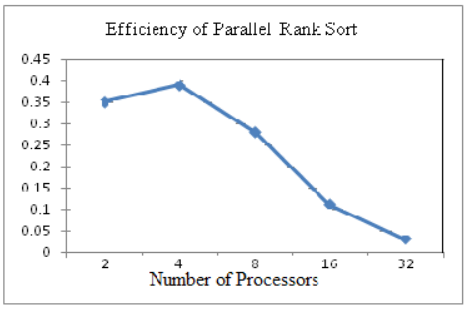

Fig. 13 Efficiency of the Parallel Rank Sort Algorithm

The limitation of the parallel rank sort is the memory it requires in order to sort its elements. Each processor needs a copy of the whole unsorted list for it to rank its portion of elements. Another memory requirement is to construct an array proportional to the unsorted list size to enable the algorithm of sorting lists with repeated elements. The parallel rank sort algorithm can be considered as a memory intensive algorithm. Figure 11 shows the total execution time for the parallel sort algorithm. When this algorithm runs on 8 processors it can improve the total execution time by a factor slightly greater than 2. However, lot of communication overheads and data transfer is required which prevents us from increasing the performance beyond this factor. Figure 12 and 13 shows speedup and efficiency.

V. Conclusion

Three parallel sorting algorithms have been developed and executed on a homogeneous cluster of workstations. The parallel algorithms implemented are the odd even transposition sorting algorithm, the parallel rank sort algorithm and the parallel merge sort algorithm. Figure shows a comparison between the three parallel sorting algorithms when sorting 2^{14} integers on 2, 4, 8, 16 and 32 processors. According to Figures it is obvious that the parallel merge sort is the fastest sorting algorithm followed by the odd-even transposition sorting algorithm then the parallel rank sorting algorithm.

Acknowledgment

The author wants to express her gratitude to Dr. Nay Min Tun for his help and advice regarding of this topic and excellent guidance, valuable suggestions and advices. We would also like to thank all teachers from Department of Information Technology at Mandalay Technological University for their valuable suggestions.

References

[1] Kalim Qureshi, "A Practical Performance Comparison of Parallel Sorting Algorithms on Homogeneous Network of Workstations"
[2] Gropp, W., Lusk, E., Skjellum, A. (1999) Using MPI: portable parallel programming with the message-passing interface, MIT, Cambridge
[3] D. Bitton, D. DeWitt, D.K. Hsiao, J. Menon, "A Taxonomy of Parallel Sorting", ACM Computing Surveys, 16,3,pp. 287-318, September 1984.
[4] E. Lusk. Programming with MPI on clusters. In 3rd IEEE International Conference on Cluster Computing (CLUSTER'01), October 2001.
[5] Mono project (2004) Open source platform basedon .NET, http://www.mono-project.com
[6] Willcock, J., et. al. (2002) Using MPI with C# and the Common language infrastructure, Technical report TR570, Indiana University, Bloomington
[7] F. Meyer auf der Heide, A Wigderson, The Complexity of Parallel Sorting, SIAM Journal of Computing, 16, 1, pp. 100-107, February1999.
[8] Gropp, W., Lusk, E., Skjellum, A. (1999) Using MPI: portable parallel programming with the message-passing interface, MIT, Cambridge.