ADAPTATION OF MULTIWAY-MERGE SORTING ALGORITHM TO MIMD ARCHITECTURES WITH AN EXPERIMENTAL STUDY

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Levent Cantürk

April, 2002

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Attila Gürsoy

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

ADAPTATION OF MULTIWAY-MERGE SORTING ALGORITHM TO MIMD ARCHITECTURES WITH AN EXPERIMENTAL STUDY

Levent Cantürk
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
April, 2002

Sorting is perhaps one of the most widely studied problems of computing. Numerous asymptotically optimal sequential algorithms have been discovered. Asymptotically optimal algorithms have been presented for varying parallel models as well. Parallel sorting algorithms have already been proposed for a variety of multiple instruction, multiple data streams (MIMD) architectures. In this thesis, we adapt the multiway-merge sorting algorithm that is originally designed for product networks, to MIMD architectures. It has good load balancing properties, modest communication needs and well performance. The multiway-merge sort algorithm requires only two all-to-all personalized communication (AAPC) and two one-to-one communications independent from the input size. In addition to evenly distributed load balancing, the algorithm requires only size of 2N/P local memory for each processor in the worst case, where N is the number of items to be sorted and P is the number of processors. We have implemented the algorithm on the PC Cluster that is established at Computer Engineering Department of Bilkent University. To compare the results we have implemented a sample sort algorithm (PSRS Parallel Sorting by Regular Sampling) by X. Liu et all and a parallel quicksort algorithm (HyperQuickSort) on the same cluster. In the experimental studies we have used three different benchmarks namely *Uniformly*, *Gaussian*, and *Zero* distributed inputs. Although the multiway-merge algorithm did not achieve better results than the other two, which are theoretically cost optimal algorithms, there are some cases that the multiway-merge algorithm outperforms the other two like in *Zero* distributed input. The results of the

experiments are reported in detail. The multiway-merge sort algorithm is not necessarily the *best* parallel sorting algorithm, but it is expected to achieve acceptable performance on a wide spectrum of MIMD architectures.

Keywords: Sorting, parallel sorting, algorithms, multiway-merge sorting, sorting in clusters.

ÖZET

ÇOK YÖNLÜ PARALEL BİRLEŞTİRME SIRALAMA ALGORİTMASININ DENEYSEL ÇALIŞMALARI İLE BİRLİKTE ÇOKLU KOMUT ÇOKLU DATA MİMARİLERİNE UYARLANMASI

Levent Cantürk
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Nisan, 2002

Elemanları sıralama problemi, hesaplamalarda muhtemelen üzerinde en çok çalışmış olan problemlerin başında gelmektedir. Bu konuda oldukça fazla optimum algoritmalar geliştirilmiştir. Bu algoritmalar birçok paralel model üzerinde denendi. Bunlar arasında tabi ki çoklu komut çoklu data (ÇKÇD) mimarileri için önerilen ve oldukça iyi çalışan algoritmalar da yer aldı. Bu çalışmamızda biz de, esasen ürün ağları için tasarlanmış çok yönlü birleştirme paralel sıralama algoritmasını ÇKÇD mimarilerine uygun hale getirdik. Çalışmamız, iş yükünün parallel makinalara dengeli dağıtılması, bilgisayarlar arasındaki iletişim yükünün azaltilması ve kendine has performans özellikleriyle oldukça başarılı bir uyarlamadır. Çok yönlü birleştirme sıralama algoritması, sıralanacak eleman sayısından bağımsız olarak sadece iki kere bütün bilgisayarlardan bütün bilgisayarlara kişisel iletişim ve iki kere de bilgisayardan bilgisayara iletişime ihtiyaç duymaktadır. Ek olarak, bu algoritma en kötü olasılıkla 2N/P kadar lokal belleğe ihtiyaç duymaktadır. Burada N sıralanacak eleman sayısını, P ise sıralamada kullanılacak işlemci sayısını temsil etmektedir. Algoritmayı Bilkent Üniversitesi Bilgisayar Mühendisliğinde kurulmuş olan dağıtık bellekli bilgisayar kümesi üzerinde programlayarak test ettik. Sonuçları karşılaştırma açısından bir tane örneklemeye dayalı paralel sıralama algoritmalarından (PSRS) bir

tane de paralel hızlısıralama algoritması örneğini (Hyperquicksort) aynı sistem üzerinde geliştirdik. Deneylerimizde girdi verilerinin dağılımlarına dayanan üç farklı kalite testi "uniformly", "Gaussian" ve "Zero" olmak üzere uyguladık. Çok yönlü birleştirme algoritması diğer iki algoritmaya göre daha iyi sonuçlar elde etmemesine rağmen, "Zero" kalite testinde olduğu gibi bazı durumlarda da diğer algoritmaları geçmiştir. Deneylerin sonuçları raporda detaylı olarak sunulmuştur. Çok yönlü birleştirme algoritması en iyi sıralama algoritması olmamasına rağmen, bir çok ÇKÇD mimarisindeki bilgisayarda çalışabilecek ve kabul edilebilir performans verebilecek bir algoritmadır.

Anahtar sözcükler: Sıralama, paralel sıralama, algoritmalar, çokyönlü-birleştirme sıralaması, bilgisayar kümelerinde sıralama.

Acknowledgement

I would like to express my special thanks and gratitude to Prof. Dr. Cevdet Aykanat, from whom I have learned a lot, due to his supervision, suggestions, and support during the past year. I would like especially thank to him for his understanding and patience in the critical moments.

I am also indebted to Assoc. Prof. Dr. Özgür Ulusoy and Assist. Prof. Dr. Attila Gürsoy for showing keen interest to the subject matter and accepting to read and review this thesis.

I would like to thank to my parents for their morale support and for many things.

I am grateful to all the honorable faculty members of the department, who actually played an important role in my life to reaching the place where I am here today.

I would like to individually thank all my colleagues and dear friends for their help and support especially to Bora Uçar and Barla Cambazloğlu.

Contents

1 Introduction ... **1**
1.1 Overview ... 1
1.2 Outline of the Thesis ... 2
2 Parallel Sorting ... **4**
2.1 Motivation ... 4
2.2 Parallel Architectures ... 5
2.2.1 PC Clusters ... 5
2.2.2 Bilkent University Beowulf PC Cluster ‘*BORG*’ ... 6
2.3 The Sorting Problem ... 8
2.4 Related Works ... 8
2.4.1 Bitonic Sort Algorithm ... 10
2.4.2 Odd-Even Merge Sort Algorithm ... 12
2.4.3 Column Sort ... 14
2.4.4 Multiway-Merge Algorithm ... 16
2.4.5 Parallel Sorting by Regular Sampling (PSRS) ... 20
2.4.6 Hypercube Quicksort ... 23
3 Adaptation of Multiway-Merge Algorithm to MIMD Architectures ... **26**
3.1 The Adapted Algorithm ... 26
3.2 A Complete Example ... 32
3.3 Complexity Analysis ... 35
3.4 Design Approach ... 38
4 Experimental Results and Analysis ... **40**
4.1 Evaluating Algorithms ... 40
4.1.1 Running Time ... 40
4.1.2 Number of Processors ... 41
4.1.3 Cost ... 42
4.1.4 Other Measures ... 42

4.2 Experimental Data 43
4.3 Comparisons of the Results 56
5 Conclusion 70
6 Appendices 79

List of Figures

2.1 The operations of steps 2 and 4 of columnsort. This figure is taken from [16]. For simplicity, this small matrix is chosen to illustrate the steps, even though its dimensions fail to obey the *Column Sort* restrictions on r and s..........................14
2.2 The operations of steps 6 and 8 of *Column Sort*. This figure is taken from [16]. .15
2.3 A complete example for column sort..16
3.1 Initial unsorted N numbers distributed to P processors..27
3.2 Logical representation of step 2 for one processor, arrows..................................28
3.3 Logical representation of step 3 for one processor, arrows indicate column wise reading order 29
4.1 Comparison of different benchmarks, while sorting on 16 nodes of *BORG with PSORT*...47
4.2 Scalibility of sorting *integers* [U] with respect to problem size, for different numbers of processors using PSORT..48
4.3 Scalability in problem size for 16 nodes 49
4.4 Scalibility of sorting 8M *integers* for different number of processors.................49
4.5 Speedup versus number of processors for different sizes of input [U].................51
4.6 Speedup versus number of processors for different sizes of input [G]..................52
4.7 Speedup versus number of processors for different sizes of input [Z]..................52
4.8 Distribution of execution time by step on 16 nodes of *BORG* 55
4.9 Distribution of execution time by step on 8 nodes of *BORG* 55
4.10 Execution time versus number of uniformly distributed [U] inputs....................60
4.11 Execution time versus number of Gaussian distributed [G] inputs.....................61
4.12 Execution time versus number of zero distributed [Z] inputs 62
4.13 Speed up versus number of uniformly distributed [U] inputs 63
4.14 Speed up versus number of Gaussian distributed [G] inputs..............................64
4.15 Speed up versus number of zero distributed [Z] inputs......................................65

List of Tables

4.1 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [U] with PSORT on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input...45

4.2 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [G] with PSORT on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input...45

4.3 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [Z] on with PSORT various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input...46

4.4 Speedup values of PSORT for different input [U] sizes on various number of processors...50

4.5 Speedup values of PSORT for different input [G] sizes on various number of processors...50

4.6 Speedup values of PSORT for different input [Z] sizes on various number of processors...50

4.7 Efficiency of PSORT for various numbers of processors on different input [U] sizes...53

4.8 Efficiency of PSORT for various numbers of processors on different input [G] sizes...53

4.9 Efficiency of PSORT for various numbers of processors on different input [Z] sizes...54

4.10 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M integers [U] with Hyperquicksort on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential Quicksort algorithm on the input...57

4.11 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M integers [G] with Hyperquicksort on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential Quicksort algorithm on the input. 57
4.12 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M integers [Z] with Hyperquicksort on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential Quicksort algorithm on the input. 58
4.13 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M integers [U] with PSRS on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential Quicksort algorithm on the input. 58
4.14 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [G] with PSRS on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input. 59
4.15 Comparison of total execution times (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [U] with PSORT, PSRS, and QSORT on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input. 60
4.16 Comparison of total execution times (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [G] with PSORT, PSRS, and QSORT on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input. 61
4.17 Comparison of total execution times (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [Z] with PSORT, PSRS, and QSORT on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input. NA means *Not Applicable* 62
4.18 Comparison of speedup values for different input [U] sizes on various number of processors for PSORT, PSRS, QSORT algorithms 63

4.19 Comparison of speedup values for different input [G] sizes on various number of processors for PSORT, PSRS, QSORT algorithms 64
4.20 Comparison of speedup values for different input [Z] sizes on various number of processors for PSORT, PSRS, QSORT algorithms 65
4.21 Comparison of efficiencies for various numbers of processors on different input [U] sizes for PSORT, PSRS, QSORT algorithms..67
4.22 Comparison of efficiencies for various numbers of processors on different input [G] sizes for PSORT, PSRS, QSORT algorithms..68
4.23 Comparison of efficiencies for various numbers of processors on different input [Z] sizes for PSORT, PSRS, QSORT algorithms 68
6.1 Data obtained from the experiments for sorting *integers* [U] on *BORG* with PSORT. 79
6.2 Row data obtained from the experiments for sorting *integers* [G] on *BORG with PSORT*..80
6.3 Row data obtained by the experiments for sorting *integers* [Z] on *BORG with PSORT*..81

List of Symbols and Abbreviations

PC : personal computer

SISD : single instruction single data stream

SIMD : single instruction multiple data stream

MISD : multiple instruction single data stream

MIMD : multiple instruction multiple data stream

MSIMD : multiple- single instruction multiple data stream

AAPC : all to all personalized communication

ACU : arithmetic control unit

O : asymptotic big OH

N : number of elements to be sorted

P : number of processors

MPI : message passing interface

PVM : parallel virtual machine

BORG : 32 node distributed memory MIMD PC-cluster of Bilkent University

P_i : ith processor

a_i : ith eelement of a sequence

$(a_0, a_1, \ldots, a_n)$: a sequence with n elements

2D : two dimensional

τ : is an upper bound on the latency of the network

σ : the time required for either injecting or receiving one word data from network.

T_{comp} : computation time

T_{comm} : communication time

T_{seq} : time of the best sequential algorithm

[U] : uniformly distributed data

[G] : Gaussian distributed data

[Z] : zero filled data

PSRS : parallel sorting with regular sampling

PSORT : multiway-merge parallel sorting algorithm

QSORT : Hyperquicksort

NA : Not Applicable

Chapter 1

1 Introduction

1.1 Overview

This thesis is devoted to the study of one particular computational problem and the various methods proposed for solving it on a parallel computer system. The chosen problem is that of *sorting a sequence of items* and is widely considered as one of the most important problems in the field of computing science. This work, therefore, is about *parallel sorting.*

Unlike conventional computers, which have more or less similar architectures, a host of different approaches for organizing parallel computers have been proposed. For the designer of parallel algorithms (i.e., problem-solving methods for parallel computers), the diversity of parallel architectures provides a very attractive domain to work in.

PC clusters are based on a recent technology and they apply supercomputer solutions to common hardware, *saving a lot of money*. The task for the parallel machine is divided among the cluster nodes and the communication among processors is performed through a local network. A PC cluster is really a parallel machine and it can be programmed with the same techniques used for supercomputers. The advantage of this solution is the extreme **cheapness of**

hardware components: with very low cost it is possible to buy a cluster with higher performance than a workstation, i.e. paying less money.

Although there has been an increasing interest in computer clusters, there are not enough efficient parallel algorithms developed for cluster systems. Therefore adaptation of parallel algorithms for MIMD architectures like cluster systems and testing their performances appears to be an interesting problem for research. This work covers research on the new algorithms for *Sorting* in the lights of the new trends in parallel computing like *PC clusters*.

We adapted a general algorithm for the problem of sorting a sequence of items on MIMD parallel computers like PC clusters. Whatever the input size, it requires only two all-to-all-personalized communication AAPC and two point-point communications. Its expected asymptotical running time is about $O(\frac{N\log(NP)}{P})$, where N is the number of items to be sorted and P is the number of processors used for sorting. What is more, we present and support the efficiency and scalability of multiway-merge algorithm with experimental results on a Beowulf PC cluster system. We have implemented one parallel sample sort algorithm and one parallel quicksort algorithm on the same cluster for comparing these three algorithms.

1.2 Outline of the Thesis

This work describes a parallel sorting algorithm for the problem of sorting a sequence of items on MIMD parallel computers with an experimental work on PC clusters. There have been various parallel algorithms proposed for the specific problem “sorting” on a variety of parallel architectures. The algorithms differ on the architectures where they are executed. It means that most of them depend on the special properties of the architectures on which they run.

In the second chapter of the thesis, a definition of the sorting problem is introduced. What is more, related work is given for a detailed understanding of the concepts underlying the algorithm we presented in the following chapter. In order to maintain a complete understanding edge, we briefly overviewed some important algorithms for parallel sorting. We tried to simplify the explanations of the algorithms and support with full examples.

In third chapter, our implementation of parallel sorting algorithm (which is the adaptation of multiway-merge algorithm to MIMD architectures) is described in details. A complete example is presented for a better understanding. Besides The complexity analysis of the adapted algorithm is also covered in this chapter. Lastly, a design approach, which is used in the implementation, is explained.

In the fourth chapter, the experimental results that are obtained by implementation of the algorithm on a PC cluster are reported. In addition, the important criteria for evaluating a parallel algorithm are overviewed to follow the theme presented in the chapter. A comparison of the multiway-merge algorithm with PSRS (Parallel Sorting by Regular Sampling) and Hyperquicksort is presented. The results are supported with appropriate graphics.

Finally, in the fifth chapter we concluded our work.

Chapter 2

2 Parallel Sorting

2.1 Motivation

With the growing number of areas in which computers are being used, there is an ever-increasing demand for more computing power than today's machines can deliver. Today's applications are required to process enormous quantities of data in reasonable amounts of time, because usage of computers is increasing dramatically in our daily life. In addition, the capacities of memories and storage devices are increased very fast. Thus, there is a need for extremely fast computers. However, it is becoming apparent that it will very soon be impossible to achieve significant increases in speed by simply using faster electronic devices, as was done in the past three decades.

Using a parallel computer is an alternative route to the attainment of very high computational speeds. In a parallel computer where there are several processing units, or processors, the problem is broken into smaller parts, each of which is solved simultaneously by a different processor. This way the solution time for a problem can be reduced dramatically by assembling hundreds or even thousands of processors. Especially when the rapidly decreasing cost of computer components is considered the attractiveness of this approach becomes more apparent.

Recently, there has been an increasing interest in computer clusters. A number of different algorithms have been described in the literature on parallel computation for sorting on MIMD computers. Therefore adaptation of parallel algorithms for MIMD architectures like cluster systems appears to be an interesting problem for research. This work covers research on the new algorithms for *Sorting* in the lights of the new trends in parallel computing like *PC clusters*.

2.2 Parallel Architectures

Using a parallel computer is an alternative route to the attainment of very high computational speeds. Unlike the case with uniprocessor computers, which generally follow the model of computation first proposed by von Neumann in the mid-1940s, several different architectures exists for parallel computers. SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data), MIMD (Multiple Instruction Multiple Data) are four main classifications of parallel architectures. More about parallel architectures could be found on various sources [8, 9, 11, 12, 13, 14, and 36].

In this part we will present a short introduction to the recent technology of PC (Server) Clusters and the *BORG,* the cluster system on which we are working.

2.2.1 PC Clusters

PC Clusters are piles of powerful PC or Alpha servers running the best available processors generally interconnected through a very high speed, low latency communication network. By working in parallel, they can provide huge amounts of computing power. Of course software has to be tuned to benefit from this architecture. Clustering technology offers by far the best price/performance ratio and can beat costly vector computers by an order of magnitude on many problem classes.

To date, supercomputer clusters can include mono or multiprocessor nodes, Intel, AMD and Alpha microprocessors, according to computing needs. High speed network can be chosen among Myrinet, SCI, Gigabit Ethernet, or simple Ethernet network, according to application and cost/performance objectives. Main features include easy-to-use administration graphical user interface, remote power on/off, remote boot of all or part of the cluster nodes, batch queuing of sequential and parallel jobs for optimal use of available power, intuitive user interface, full execution environment for MPI and OpenMP compliant parallel programs, performance monitoring tools, debuggers.

PC clusters are based on a recent technology and they apply supercomputer solutions to common hardware, *saving a lot of money*. The task for the parallel machine is divided among the cluster nodes and the communication among processors is performed through a local network.

A PC cluster is really a parallel machine and it can be programmed with the same techniques used for supercomputers. The advantage of this solution is the extreme **cheapness of hardware components**: with very low cost it is possible to buy a cluster with higher performance than a workstation, paying less money. Then a cluster is very scalable, and you can set up it with few PCs up to hundreds of nodes, building a machine comparable with medium-low parallel supercomputers. Performance being equal with supercomputers and workstations, **PC clusters can reach price ratios up to 10-15 times lower**.

2.2.2 Bilkent University Beowulf PC Cluster ‘*BORG*’

In this section, ‘*BORG’* computer system is introduced. Because of increasing interest on cluster systems in Parallel Computing encouraged us to test the multiway-merge sorting algorithm on *BORG*. This short introduction will help better understanding and interpretation of the results in Chapter 4.

BORG, the cluster in Bilkent University, is made up of a group of *personal computers interconnected by a non-blocking 100 Mbit switch*. The nodes of the cluster have no monitor, neither keyboard, but they have powerful processors and good RAM memory. All PCs in the cluster are set up with ***Linux*** operative system and some standard tools for parallel programming (PVM [26] and MPI [23] libraries), also used on supercomputers.

As a base model Beowulf is applied in *BORG*. Beowulf is a kind of high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system like Linux, interconnected by a private high-speed network. It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks. The nodes in the cluster don't sit on people's desks; they are dedicated to running cluster jobs. It is usually connected to the outside world through only a single node. More information about Beowulf can be found in [25].

Specifically *BORG* is a Pentium-based pile-of-PCs of 32 machines, each with the following:

- GenuineIntel 400 MHz Pentium II CPU with 512KB cache size
- 128 MB SDRAM
- 13 GB IDE disk
- 100 Mbit Ethernet cards
- Debian GNU/Linux woody distribution (3.0)

The experimental results in Chapter 4 were obtained by executing the algorithms on this system.

2.3 The Sorting Problem

Sorting is probably the most well studied problem in computing science due to practical and theoretical reasons. It is often said that 25-50% of all the work performed by computers consists of sorting data. The problem is also of great theoretical appeal, and its study has generated a significant amount of interesting concepts and beautiful mathematics. A formal definition is given as follows in [14].

Definition 1.1 The elements of set A are said to satisfy a *linear order* < if and only if

(1) for any two elements a and b of A, either $a < b$, $a = b$, or $b < a$; and

(2) for any three elements a, b, and c of A, if $a < b$ and $b < c$, then $a < c$.

The linear order < is usually read “precedes”.

Definition 1.2 Given a sequence $S = \{x_1, x_2, \ldots, x_n\}$ of n items on which a linear order is defined, the purpose of *sorting* is to arrange the elements of S into a new sequence $S' = \{ x_1', x_2', \ldots, x_n'\}$ such that $x_i' < x_{i+1}'$ for $i = 1, 2, \ldots, n - 1$.

2.4 Related Works

Sorting is used in many applications and many of the algorithms in computer science depend on sorting. They require sorted data, since they are easier to manipulate than randomly ordered data. Simply if computer world would be thought as a world of zeros and ones, the only distinguished property of these are their orderings. Hence rearrangement of the ordering, sorting, has an important role in computer science. There are many asymptotically optimal algorithms found in this area. After parallel computing takes attention of scientists, parallelism of sorting was an interesting topic. Therefore today there are lots of parallel sorting algorithms offered for many varieties of parallel architectures [27]. Detailed information could be obtained from [14] which is a summary of the known parallel sorting algorithms proposed till 1985 and

following sections for later works. Since most of the famous sorting algorithms are well known, only the algorithms related with multiway-merge algorithm are covered in this section.

Most of the parallel sorting algorithms are based on the ideas in [2] of Batcher. The ideas in [2] form a basis even for today's works about parallel sorting. As mentioned in [1], algorithms that take the intuition from the ideas presented by Batcher applied to variety of parallel architectures like the shuffle-exchange network [30], the grid [10, 31], the cube-connected cycles [32], and the mesh of trees [33]. What is more, PSRS from [19, 20], and a series of works by D. Bader and et al [7] are considerable studies that could able to achieve the optimum algorithms. We can split these algorithms in two groups as Li and Sevick [38] did; the single–step algorithms and the multi-step algorithms. In the former one data, is moved only once between processors. PSRS [19, 20], sample sort [7, 39, 41] and parallel sorting by overpartitioning [38] could be classified in this group. Irregular communication requirements and difficulties in load balancing are disadvantages of the algorithms that fall into this group. The latter one -multi-step algorithms- may require multiple rounds of communication in order to obtain better load balancing. Bitonic sort [2], Column Sort [15, 16], SmoothSort [40], Hyperquicksort [18, 42], B-Flashsort [43], and Brent's sort [44] fall into the second category. We have selected one sample algorithm from each group and implemented them in our cluster system *BORG* with MPI to compare with the multiway-merge algorithm.

In order to summarize famous algorithms in parallel sorting and to follow the work presented here much better, we reviewed related sorting algorithms in literature in the next following sections with a complete example for each. Batchers [2] ***Bitonic Sort*** and ***Odd-Even Merge Sort*** are important for the concept of parallel sorting. Leightons [15, 16] ***Column Sort*** is also crucial for understanding the ***Multiway-Merge Sort*** [1], and also for the adapted algorithm. ***PSRS*** and ***Hyperquicksort*** are covered since they are the selected algorithms for comparison.

2.4.1 Bitonic Sort Algorithm

One of the two efficient sorting algorithms of Batcher [2] is *Bitonic Sorting*. The algorithm sorts some special kind of sequence called *bitonic sequences*. A bitonic sequence [18, 1, 14] is a sequence of elements ($a_1, a_2, \ldots, a_n$) which satisfies either following properties:

(1) there exists an index i, $1 \leq i \leq n$, such that ($a_1, a_2, \ldots, a_i$) is monotonically increasing and ($a_{i+1}, a_{i+2}, \ldots, a_n$) is monotonically decreasing, or

(2) there exists a cyclic shift of indices so that (1) is satisfied, i.e. any rotation of a sequence which satisfies (1).

For instance, (2, 3, 5, 7, 9, 8, 4, 0) is a bitonic sequence, since it first increases and then decreases. On the other hand (1, 4, 8, 6, 5, 9) is not a bitonic sequence, because the last element 9 violates the property (1). (7, 8, 3, 1, 5) is also a bitonic sequence, because it is a cyclic shift of (5, 7, 8, 3, 1) which first increases and then decreases.

A monotonically increasing sequence can always be obtained from a bitonic sequence by applying recursively the bitonic split algorithm. A bitonic split is defined as the operation of splitting a bitonic sequence into two bitonic sequences. The **bitonic split** algorithm is as follows.

Consider $A = (a_1, a_2, \ldots, a_n)$ as a bitonic sequence such that $a_1 \leq a_2 \leq \ldots, \leq a_{n/2-1}$ and $a_{n/2} \geq a_{n/2+1} \geq \ldots, \geq a_n$ then the following subsequences A_1 and A_2 will give two bitonic subsequences of A.

A_1 = (min[$a_1, a_{n/2}$], min[$a_2, a_{n/2+1}$],…, min[$a_{n/2-1}, a_n$]) and

A_2 = (max[$a_1, a_{n/2}$], max[$a_2, a_{n/2+1}$],…, max[$a_{n/2-1}, a_n$]).

Sorting a bitonic sequence using bitonic splits is called ***bitonic merge.***

EXAMPLE: Simulation of a merge operation on bitonic sequence of n = 16 element, that requires $\log_2 16$ bitonic splits.

Initial sequence:	4, 6, 8, 9, 10, 11, 14, 16, 80, 70, 60, 50, 40, 30, 20, 3
Split 1:	4, 6, 8, 9, 10, 11, 14, 3, 80, 70, 60, 50, 40, 30, 20, 16
Split 2:	4, 6, 8, 3, 10, 11, 14, 9, 40, 30, 20, 16, 80, 70, 60, 50
Split 3:	4, 3, 8, 6, 10, 9, 14, 11, 20, 16, 40, 30, 60, 50, 80, 70
Split 4:	3, 4, 6, 8, 9, 10, 11, 14, 16, 20, 30, 40, 50, 60, 70, 80

Sorting a sequence of length n elements by repeatedly merging bitonic sequences of increasing length is called ***bitonic sort***. Since any subsequence with two elements is a bitonic sequence, the algorithm starts from the subsequences of length two, and then length of four, and goes like that. In each intermediate stage the adjacent bitonic sequences are merged in increasing and decreasing order respectively. The sequences obtained in the intermediate steps are also bitonic sequences, because concatenations of increasing and decreasing sequences are also bitonic sequences. The Bitonic Sorting Network was the first network that is capable of sorting n elements in $O(\log^2 n)$ time.

EXAMPLE: Simulation of *Bitonic Sorting* on a sequence of n = 8 elements. The intermediate merges are omitted for simplicity; however the final bitonic merge is simulated in details for clarity.

X : increasing order

X: decreasing order

Initial sequence:	0, 3, 2, 6, 4, 1, 5, 7
Bitonic Merging sequences of length two:	**0, 3**, *2, 6*, **4, 1**, *5, 7*
Result:	**0, 3**, *6, 2*, **1, 4**, *7, 5*
Bitonic Merging sequences of length four:	**0, 3, 6, 2**, *1, 4, 7, 5*
Result:	**0, 2, 3, 6**, *7, 5, 4, 1*
Bitonic Merging sequences of length eight:	**0, 2, 3, 6, 7, 5, 4, 1**

Final merge in details:	**0, 2, 3, 6, 7, 5, 4, 1**
Split 1.	**0, 2, 3, 1, 7, 5, 4, 6**
Split 2.	**0, 1, 3, 2, 4, 5, 7, 6**
Split 3.	**0, 1, 2, 3, 4, 5, 6, 7**

Sorting algorithms based on this method are called *bitonic sorters* and there are many papers about generalization of bitonic sorters in the literature. [3], [4], [5], [34].

2.4.2 Odd-Even Merge Sort Algorithm

Odd-Even Merge Sort [13, 10] is one of the oldest and most famous algorithms in parallel computing. Let A be a sequence of n keys to be sorted. The odd-even merge sort algorithm [2] applies the odd-even merge algorithm repeatedly to merge two sequences at a time. Initially it forms $n/2$ sorted sequences of length two each. Next, it merges two sequences at a time so that at the end $n/4$ sorted sequences of length four each will remain. This process of merging continued until only two sequences of length $n/2$ each are left. Finally, these two sequences are merged. The odd-even merge algorithm can be ruled simply as follows.

Step 1. Assume we want to merge two sorted sequences A and B, where $A= a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_n$. In the first step, two sequences are partitioned in to two subsequences like $A_{odd} = a_1, a_3, a_5, \ldots, a_{n-1}$ and $A_{even} = a_2, a_4, a_6, \ldots, a_n$. Similarly B is partitioned into two sequences $B_{odd} = b_1, b_3, b_5, \ldots, b_{n-1}$ and $B_{even} = b_2, b_4, b_6, \ldots, b_n$.

Step 2. A_{odd} and B_{even} are merged and A_{even} and B_{odd} are merged recursively. Let call the results as $O= o_1, o_2, \ldots, o_n$ and $E = e_1, e_2, \ldots, e_n$, respectively.

Step 3. Combine the sequences O and E and generate a new sequence C by taking one element from O and one element from E until all elements are exhausted. Hence C will be $o_1, e_1, o_2, e_2, \ldots, o_n, e_n$.

Step 4. Finally, starting from element 2, sorting the subsequences of length two successively in *C*, will give the result of merging the initial sequences *A* and *B*.

Although [15] uses this algorithm, there are many variations of the odd-even merge in the literature. For example, in [17] step 2 is changed as recursively merging A_{odd} with B_{odd} and A_{even} with B_{even} to yield corresponding results *O* and *E*. In step 4, they sort the subsequences of length two successively in *C* starting from the second element.

EXAMPLE: Simulation of *Odd-Even Merge* algorithm. Let us consider the problem of merging two sorted sequences *A* and *B*, where *A* = 0, 2, 3, 6, 8, 10, 12,13 and *B* = 1, 4, 5, 7, 9, 11, 14, 15

Step 1. A_{odd} = 0, 3, 8, 12 and A_{even} = 2, 6, 10, 13
B_{odd} = 1, 5, 9, 14 and B_{even} = 4, 7, 11, 15
Step 2. *O* = 0, 3, 4, 7, 8, 11, 12, 15 and *E* = 1, 2, 5, 6, 9, 10, 13, 14
Step 3. *C* = 0, 1, 3 , 2, 4, 5, 7, 6, 8, 9, 11, 10, 12, 13, 15, 14
Step 4. *C* = 0, 1, 3 , 2, 4, 5, 7, 6, 8, 9, 11, 10, 12, 13, 15, 14
Result = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

EXAMPLE: Simulation of *Odd-Even Merge Sort* on a sequence of *n* = 8 elements.

Initial sequence:	1, 3, 7, 6, 2, 0, 4, 5
forms *n*/2 sorted sequences of length 2:	1, 3, 6, 7, 0, 2, 4, 5
odd-even merge of successive sequences:	1, 3, 6, 7, 0, 2, 4, 5
finally merge the 2 last sequences:	0, 1, 2, 3, 4, 5, 6, 7

Like the Bitonic Sort, Odd-Even Merge Sorting Network was one of the first network that is capable of sorting *n* elements in $O(\log^2 n)$ time. Although it is one of the oldest parallel algorithms, it is still one of the most widely used and important algorithms in parallel sorting.

2.4.3 Column Sort

In this section, we review the Leightons Column Sort [15], which plays an important role in the adapted algorithm, and also in the following *Multiway-merge Algorithm.*

As in many algorithms, there are many variations of Column Sort in the literature. The simplest version of Column Sort presented by Leighton in [15] is a seven phase algorithm that sorts *N* items into column major order in an *r* x *s* matrix. Steps 1, 3, 5, and 7 are all the same: sort each column individually. The only exception is the sorting in Step 5. In Step 5, adjacent columns are sorted in reverse order, however in other steps sorting is in the same order (smallest-first) for all columns. Each of Steps 2, 4, and 6 permutes the matrix entries. In Step 2, the matrix is transposed by picking up the items in column-major order and setting them down in row-major order (preserving the *r* x *s* shape, illustrated in Figure 2.1). As Leighton stated, the reverse permutation in the Step 2 is applied, picking up items in row-major order and setting them down in column-major order. Step 6 consists of two steps of odd-even transposition sort to each row. Column Sort is always sorts *N* items into column-major order provided that $r \geq s^2$.

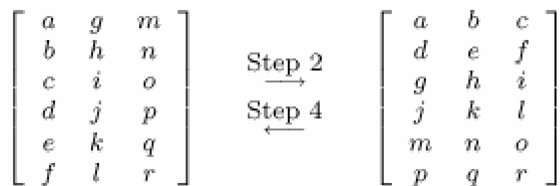


Figure 2.1 The operations of Steps 2 and 4 of columnsort. This figure is taken from [16]. For simplicity, this small matrix is chosen to illustrate the steps, even though its dimensions fail to obey the *Column Sort* restrictions on r and s.

As in [16] and [37] one other famous version of column sort is an eight phase algorithm. Steps 1, 3, 5, 7 are similar with the previous, just sorting the columns individually in smallest-first order without any exception i.e. even in Step 5 the ordering is same. Step 2 and 4 are also same as in previous. In Step 6, each column is shifted down by $r/2$ positions. This shift operation empties the first $r/2$ entries of the leftmost column, which are filled with keys of $-\infty$, and it creates a new column on the right, the bottommost r/2 entries of which are filled with keys of ∞. In another words, top half of each column is shifted into the bottom half of that column, and the bottom half of each column is shifted into the top half of the next column. In Step 8, the reverse permutation of Step 6 is applied. Step 6 and 8 are illustrated in Figure 2.2.

$$\begin{bmatrix} a & g & m \\ b & h & n \\ c & i & o \\ d & j & p \\ e & k & q \\ f & l & r \end{bmatrix} \quad \begin{matrix} \text{Step 6} \\ \longrightarrow \\ \text{Step 8} \\ \longleftarrow \end{matrix} \quad \begin{bmatrix} -\infty & d & j & p \\ -\infty & e & k & q \\ -\infty & f & l & r \\ a & g & m & \infty \\ b & h & n & \infty \\ c & i & o & \infty \end{bmatrix}$$

Figure 2.2 The operations of Steps 6 and 8 of *Column Sort*. This figure is taken from [16].

EXAMPLE: Simulation of *Column Sort* on a sequence of $N = 8$ elements with $r = 4$ and $s = 2$.

Initial sequence: 3, 2, 1, 5, 6, 0, 7, 4

After writing the elements in *r* x *s,* 4 x2 array:

3	2
1	5
6	0
7	4

Initial array

1	0
3	2
6	4
7	5

Step 1. Sort columns

1	3
6	7
0	2
4	5

Step 2. Transpose

0	2
1	3
4	5
6	7

Step 3. Sort columns

0	4
2	5
1	6
3	7

Step 4. Transpose

0	7
1	6
2	5
3	4

Step 5. Sort columns

0	6
1	7
2	4
3	5

Step 6.1. Odd-even transp.

0	6
1	4
2	7
3	5

Step 6.2. Odd-even transp.

0	4
1	5
2	6
3	7

Step 7. Sort columns

Figure 2.3 A complete example for column sort

2.4.4 Multiway-Merge Algorithm

In this section, we will present the previous related works with the topic of *sorting*, especially the multiway- merge sorting algorithm that in fact gives the spirit of this thesis, in [1].

In [1] the well known odd-even merge sorting algorithm, originally due to Batcher[2] is generalized and its application on product networks are presented. In addition, they developed a new multiway-merge algorithm that merges several sorted sequences into a single sorted sequence. By using their multiway-merge algorithm they obtained a new parallel sorting algorithm that they claimed probably the best deterministic algorithm, which can be found in terms of the low asymptotic complexity with a small constant in product networks. Therefore, adaptation of that algorithm for MIMD architectures was an interesting research. In [1], the application of the multiway-merge sorting algorithm to some special product networks is covered. Efe and Fernandez give the asymptotic analysis of the algorithm for *Grid, Mesh-Connected Trees (MCT), Hypercube, Petersen Cube, Product of de Brujin,* and *Shuffle-Exchange Networks* in [1]. For *Grid* and *Mesh-Connected Trees* they obtained a bound O (N), which is optimal. For other product networks they also obtained quite optimal asymptotical bounds.

In the sorting algorithm [1], they applied the proposed multiway-merge operation recursively to the sorted subsequences. In the paper, the multiway-merge algorithm is defined as follows.

Assume we consider to merge N sorted subsequences, $A_i = (a_0, a_1, \ldots, a_{m-1})$, for $i = 0, 1, \ldots, N-1$, into a large sorted sequence N'.

Step 1. Initially, the algorithm requires the distribution of each sorted sequence A_i among N sorted subsequences $B_{i,v}$, for $i = 0, 1, \ldots, N-1$ and $v = 0, 1, \ldots, N-1$. This is equivalent to writing the keys of each A_i on a m/N by N array in snake order and then reading the keys column-wise to obtain each $B_{i,v}$.

Step 2. The N subsequences $B_{i,v}$ found in column v are merged into a single sorted sequence C_v for $v = 0, 1, \ldots, N-1$. This step is handled in parallel for all columns by a recursive call to the multiway-merge itself, if the total number of keys in the column

m is at least N^3. For the columns whose length is less than N^3 like N^2 a sorting algorithm is used instead of a recursive call to merge.

Step 3. The sequences C_v for $v = 0, 1, \ldots, N$-1 are interleaved into a single sequence $D = (d_0, d_1, \ldots, d_{mN-1})$, by reading the columns C_v*'s* in *row-major* order starting from the top row.

Step 4. To clean the '*dirty area'* (i.e. unsorted portion), D is divided into m/N subsequences of N^2 consecutive keys each. These subsequences are called as E_z, where $z = 0, 1, \ldots, m/N - 1$. Next, these subsequences are sorted in alternate orders, that is, sequence is sorted in nondecreasing order for even z and in nonincreasing order for odd z. Then two step odd-even transposition is applied between the sorted sequences in the vertical direction. In the first transposition, the elements in the rows z and $z + 1$ are compared and the smallest is stored in row z, while the largest is stored in the z+1th row for even z values. The second transposition is application of the same to the odd z values. Lastly, the final subsequences E_z's are sorted in alternate orders as in previous one. The whole sorted sequence *N'* is just concatenation of the sequences E_z's in snake-order.

Lemma 1[1]: *"When sorting an input sequence of zeros and ones, the sequence D obtained after the completion of Step 3 is sorted except for a dirty area which is never larger than* N^2*."*

Proof of this Lemma is also presented in [1].

EXAMPLE: Simulation of multiway-merge algorithm. Assume we merge N=3 sorted subsequences , $A_i = (a_0, a_1, \ldots, a_{m-1})$, for $i = 0, 1, \ldots, 2$ and $m = 9$ into a large sorted sequence *N'*.

Initial sequences:

A_0 = 0, 2, 5, 5, 6, 7, 8, 9, 9

A_1 = 0, 2, 3, 6, 6, 7, 7, 8, 9

A_2 = 1, 2, 4, 5, 6, 6, 7, 8, 9

After Step1:

B_0 = <u>0, 7, 8</u>, <u>2, 6, 9</u>, <u>5, 5, 9</u>

B_1 = <u>0, 7, 7</u>, <u>2, 6, 8</u>, <u>3, 6, 9</u>

B_2 = <u>1, 6, 7</u>, <u>2, 6, 8</u>, <u>4, 5, 9</u>

After Step 2:

C_0	C_1	C_2
0	2	3
0	2	4
1	2	5
6	6	5
7	6	5
7	6	6
7	8	9
7	8	9
8	9	9

After Step 3: D : 0, 2, 3, 0, 2, 4, 1, 2, 5, 6, 6, 5, 7, 6, 5, 7, 6, 6, 7, 8, 9, 7, 8, 9, 8, 9, 9

For Step 4:

E_0 : 0, 2, 3, 0, 2, 4, 1, 2, 5

E_1 : 6, 6, 5, 7, 6, 5, 7, 6, 6

E_2 : 7, 8, 9, 7, 8, 9, 8, 9, 9

After sort alternate order

E_0 : 0, 0, 1, 2, 2, 2, 3, 4, 5

E_1 : 7, 7, 6, 6, 6, 6, 6, 5, 5

E_2 : 7, 7, 8, 8, 8, 9, 9, 9, 9

After odd-even transitions and sort alternate order

E_0 : 0, 0, 1, 1, 2, 2, 3, 4, 5

E_1 : 7, 7, 6, 6, 6, 6, 6, 5, 5

E_2 : 7, 7, 8, 8, 8, 9, 9, 9, 9

So, N' = 0, 0, 1, 1, 2, 2, 3, 4, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 9, 9, 9, 9

2.4.5 Parallel Sorting by Regular Sampling (PSRS)

One of the famous algorithm among the recent parallel *sample sorting* algorithms is parallel sorting by regular sampling offered by X.Li et al [19, 20] in 1996. This algorithm is suitable for a diverse range of MIMD architectures. The time complexity of PSRS is asymptotic to $O(\frac{n}{p}\log n)$ when $n \geq p^3$, which is cost optimal [20] . It has a good theoretical upperbound on the worst case load balancing among other sample sort algorithms. If we assume there is no duplicate keys, it has proven that in PSRS no processor has to work on more than $2n/p$ data elements if $n \geq p^3$.

The algorithm consists of four phases; a sequential sort, a load balancing phase, a data exchange and a parallel merge. In order to sort n numbers (with indices 1, 2, 3, …, n) the algorithm uses p processors (1, 2, 3, …, p), so as in previous algorithms initially the data n is distributed over p processors. Basically we assume each processor has its random portion of the data already stored within them.

Phase 1. In parallel, each processor sorts their contiguous block of n/p items that is assigned to them. A sequential quicksort or merge sort can be used here. Expected runtime of the quicksort algorithm is $O(n\log n)$, but may have a worst case of $O(n^2)$. If this is a problem, an algorithm with a worst case of $O(n\log n)$, such as merge sort, can be used. After local sorting, processors select the samples that represent the locally sorted blocks. Here in this part, processors select the elements at local indices 1, $n/p+1$, $2n/p+1$, …, $(p-1)n/p+1$, to form the samples. These p elements are called *regular samples* of the local elements and represent the value distribution at each processor. Therefore collection of these p elements from p processors will give the *regular sample* of the initial n numbers to be sorted. This process is called *regular sampling* load balancing heuristic.

Phase 2. This phase consists of pivot finding and local partitioning according to the pivots. In order to find pivots each processor sends the local regular samples which are potential candidates of the pivots, to a specific processor. The designated processor sorts the regular samples and selects the elements with indices $p + \lfloor p/2 \rfloor$, $2p + \lfloor p/2 \rfloor$, $3p + \lfloor p/2 \rfloor$, …, $(p\text{-}1) + \lfloor p/2 \rfloor$. These selected ($p$-1) elements forms the pivots. These pivots are distributed to each processor. Upon receiving the processors, each processor forms p partitions (s_1, s_2, s_3, …,s_p) from their sorted local blocks according to the pivots.

Phase 3. Processors apply the total exchange algorithm for their p partitions. In other words, each processor i keeps the ith partition s_i itself and assigns the j partitition s_j to the jth processor. As an example, processor 3 keeps the 3rd partition itself and gathers all the 3rd partitions of other p-1 processors, while sending its remaining partitions to the appropriate processors.

Phase 4. In this phase, each processor merges its p partitions in parallel. Since the partitions are already sorted, P-way merge algorithm will give a whole sorted sequence. After completion of phase 4, any element in processor i is greater than any element in processor j where $i > j$ and within each processor n/p elements are sorted among themselves. So the concatenation of all local lists will give the final n element sorted list.

The implementation details and experimental results are explained in later chapters.

EXAMPLE: Simulation of *PSRS* on a sequence of n = 27 elements with p = 3 processors [42].

Initial unsorted sequence:

8, 23, 15, 3, 12, 22, 6, 21, 0, 9, 26, 11, 4, 20, 14, 2, 16, 24, 19, 18, 13, 7, 5, 1, 17, 25, 10

The elements are distributed to the processors evenly:

P_1 = 8, 23, 15, 3, 12, 22, 6, 21, 0

P_2 = 9, 26, 11, 4, 20, 14, 2, 16, 24

P_3 = 19, 18, 13, 7, 5, 1, 17, 25, 10

PHASE 1

Each processor sort its local elements (n/p = 27/3=9) with a sequential sorting algorithm. Then they select, in parallel, their local regular samples.

P_1 = ***0***, 3, 6, ***8***, 12, 15, ***21***, 22, 23 *Local regular samples:* 0, 8, 21

P_2 = ***2***, 4, 9, ***11***, 14, 16, ***20***, 24, 26 *Local regular samples:* 2, 11, 20

P_3 = ***1***, 5, 7, ***10***, 13, 17, ***18***, 19, 25 *Local regular samples:* 1, 10, 18

PHASE 2

Local regular samples are gathered and sorted (p^2 = 9 elements). Two (p-1) pivots are selected and distributed to the processors. Each processor generates three partitions according to the pivots.

Gathered Regular Sample: 0, 8, 21, 2, 11, 20, 1, 10, 18

Sorted Regular Sample : ***0***, 1, 2, ***8***, 10, 11, ***18***, 20, 21

Pivots : ***8***, ***18***

P_1 = 0, 3, 6, 8,	12, 15,	21, 22, 23
P_2 = 2, 4,	9, 11, 14, 16,	20, 24, 26
P_3 = 1, 5, 7,	10, 13, 17, 18,	19, 25

PHASE 3

Total exchange algorithm on the partitions.

P_1 = 0, 3, 6, 8,	2, 4,	1, 5, 7
P_2 = 12, 15,	9, 11, 14, 16,	10, 13, 17, 18
P_3 = 21, 22, 23,	20, 24, 26,	19, 25

PHASE 4

Each processor merges its new *p* partitions after total exchange.

P_1 = 0, 1, 2, 3, 4, 5, 6, 7, 8

P_2 = 9, 10, 11, 12, 13, 14, 15, 16, 17, 18

P_3 = 19, 20, 21, 22, 23, 24, 25, 26

Final sorted array:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26

2.4.6 Hypercube Quicksort

Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead and optimal average complexity. Quicksort is often the best practical choice for sorting because its very efficient on the average that is, its expected running time is $\Theta(n\log n)$ for sorting n numbers. Although its worst case running time is $\Theta(n^2)$, it also uses the advantages of inplace algorithms and performs well even in virtual memory environments. Of course, there are variety of parallel versions of such a famous algorithm in the literature [39, 46, 47, 48]. They are similar in the main idea, but differ in the architectures they run, selection of the pivots, or some parallel specific concepts like load balancing etc. In this section we present one of the parallel formulations of the quicksort algorithm [18, 42].

As it is known, Quicksort is a diviede&conquer paradigm algorithm. The simple algorithm to sort an array $A[p, r]$ given in [45] is as follows:

Divide : The array $A[p,r]$ is partitioned into two nonempty sub arrays $A[p,q]$ and $A[q+1, r]$, such that each element in the former is less than or equal to each element in the latter. The index q is computed as a part of this partitioning process.

Conquer: The two subarrays $A[p, q]$ and $A[q+1, r]$ are sorted recursively by calling Quicksort.

Combine: The entire array $A[p, r]$ is sorted, since the subarrays are sorted in place, so work for combine step.

One of the basic ways of parallelizing of Quicksort is running of the recursive calls in parallel processors. Since sorting of the subarrays are two completely independent subproblems, they can be solved in parallel. Three distinct parallel formulations of Quicksort : one for a CRCW PRAM, one for a hypercube, and one for a mesh is presented in [18]. In this section, we are briefly focused on HyperQuick[1] sort algorithm which we have implemented in *BORG* for comparison purposes.

Hypercube Quicksort, as it is understood from its name, uses advantage of topological properties of a hypercube. The hyperquicksort algorithm works as follows. Let n be the number of elements to be sorted and $p=2^d$ be the number of processors in a d-dimensional hypercube. Initially we assume, as it is a convention in most algorithms in this work, the n numbers are evenly distributed to each processor. So each processor gets a block of n/p elements. At the end, elements in processor i is less than or equal to the elements in processor j where $i < j$. The algorithm starts by selecting a pivot element, which is broadcast to all processors. After receiving the pivot, each processor partitions its local elements into two blocks according to the pivot. All the elements smaller than the pivot are stored in one part and all the elements larger than the pivot are kept in other sub block. A d-dimensional hypercube can be decomposed into two (d-1) dimensional subcubes such that each processor in one subcube is connected a processor in other subcube. The corresponding processors could separate the subcubes according to the pivot by keeping smaller elements in one (d-1) subcube and sending the larger elements to the processors in the other subcube. Thus the processors connected along the dth communication link exchange appropriate blocks so that one retains elements smaller than the pivot and the other retains elements larger than the pivot. The proposed algorithm in [18] for this communication is, each processor with a 0 in the dth bit (the most significant bit) position of the binary representation of its processor label retains the smaller elements, and each processor with a 1 in the dth bit retains the larger elements. After

---

[1] HyperQuicksort and Hypercube QuicksortQuicksort were used interchangeble.

this step, each processor in the (d-1)-dimensional hypercube whose dth label bit is 0 will have elements smaller than the pivot and each processor in the other (d-1)-dimensional hypercube will have elements larger than the pivot. This procedure is performed recursively in each subcube, splitting the subcubes further. After d such splits – one along each dimension- the sequence is sorted with respect to the global ordering imposed on the processors. Finally, each processor sorts its local elements by using sequential Quicksort.

One important point in this algorithm is the selection of the pivot, because a bad selection of the pivot will yield poor performance. Since at each recursive step the elements are distributed according to the pivot, if the pivot does not split almost equally the elements, then this will result with a poor load balancing. What is more, the performance will degrade further at each recursive step. For example during the ith split partitioning a sequence among two subcubes with a bad pivot may cause the elements in one subcube more than the others (load imbalance). There are numerous ways of selecting the pivot. It can be the first element of a randomly selected processor or average of the average of the elements in the processors, or average of a randomly picked sample and etc. The important point here is whatever the method is; the pivot should divide almost equally the numbers to be sorted. A good approximation for a good pivot is selection of the medians at each processor for each subcube and then taking the average or median of them again. Thus the median for each processor will give a good approximation for the elements in that processor and collection of the medians of each processor will give a good sample space for that subcube in order to divide the elements evenly.

As stated in [18], the hypercube formulation of Quicksort is depends on the pivot selection. If pivot selection is good, then its scalability is relatively good. If the pivot selection is bad (worst case) then it has an exponential isoefficiency function. On the other hand, mesh formulation of Quicksort has an exponential isoefficiency function and is practical only for small values of p.

Chapter 3

3 Adaptation of Multiway-Merge Algorithm to MIMD Architectures

3.1 The Adapted Algorithm

This section develops the basic steps of the multiway-merge sorting algorithm. According to the problem definition of the sorting, the multiway-merge algorithm is expected to rearrange the sequence of keys (for the clarity, keys are assumed to be integers, but not necessarily) such that the resulting sequence follows the definition of a sorted sequence. As explained in [1] and [6], if an algorithm is able to sort any sequence of zeros and ones and based on compare-exchange operations, it can sort any items.

A *sorted sequence* [1] is defined as a sequence of keys (a_0, a_1, …, a_{n-1}) such that $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$. For clarity, let's assume P_i denotes the ith processor and A_i denotes the sequence A stored in the processor i.

Initially we have N numbers to be sorted, and P processors available. We will assume N/P is some power of P, so each processor gets some power of P^{k-1} elements where $N= P^k$ and $k > 2$. Like in similar works, the number of processors P is a power of two. To start the algorithm, it is supposed that N/P numbers are distributed evenly to the processors and stored as a sequence $A_i(a_0, a_1, \ldots, a_{N/P-1})$ in each processor. The output consists of the elements in non-descending order arranged amongst the processors so that the elements at each processor are in sorted order and no element at processor P_i is greater than any element at processor P_j, for all $i < j$.

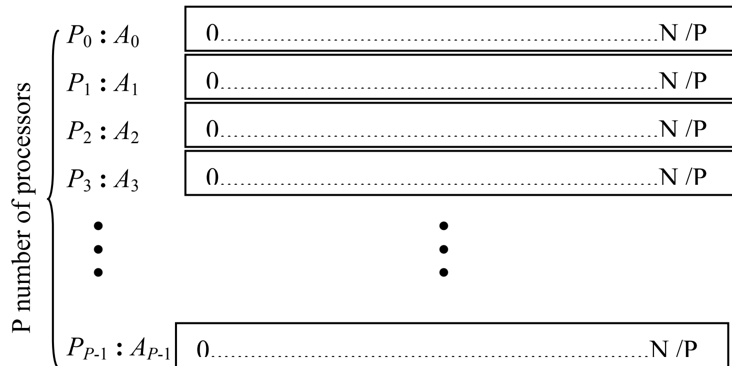


Figure 3.1 Initial unsorted N numbers distributed to P processors

Step 1. Each P processor sorts its sequence A containing N/P numbers in parallel, using a sequential sorting algorithm like Quicksort. Here Quicksort is preferred due to its storage requirements since Quicksort is an inplace algorithm and performs very well if the whole array resides on the memory. Therefore this step requires $\Theta(N/P \log(N/P))$ expected running time. On the other hand, if memory is not a concern than usage of merge sort offers an O $(N/P \log(N/P))$ as an upper bound for the worst case.

Step 2. Each P processor puts the sorted A containing N/P numbers into a local 2D array N/P^2 by $(N/P^2 \times P)$ in row-major or snake order. [1] Used snake order in the algorithm, because the orderings in snake order is more suitable for the structure of product networks. For our case, since we have two options that are

feasible, in the rest of the algorithm, we preferred the row-major order. This step is illustrated in Figure 3.2 below. In the implementation of the algorithm, this should not be interpreted to imply the physical organization of data in a two dimensional array. Actually, step 2 and 3 could be handled by just rearrangement of indices of a one dimensional array.

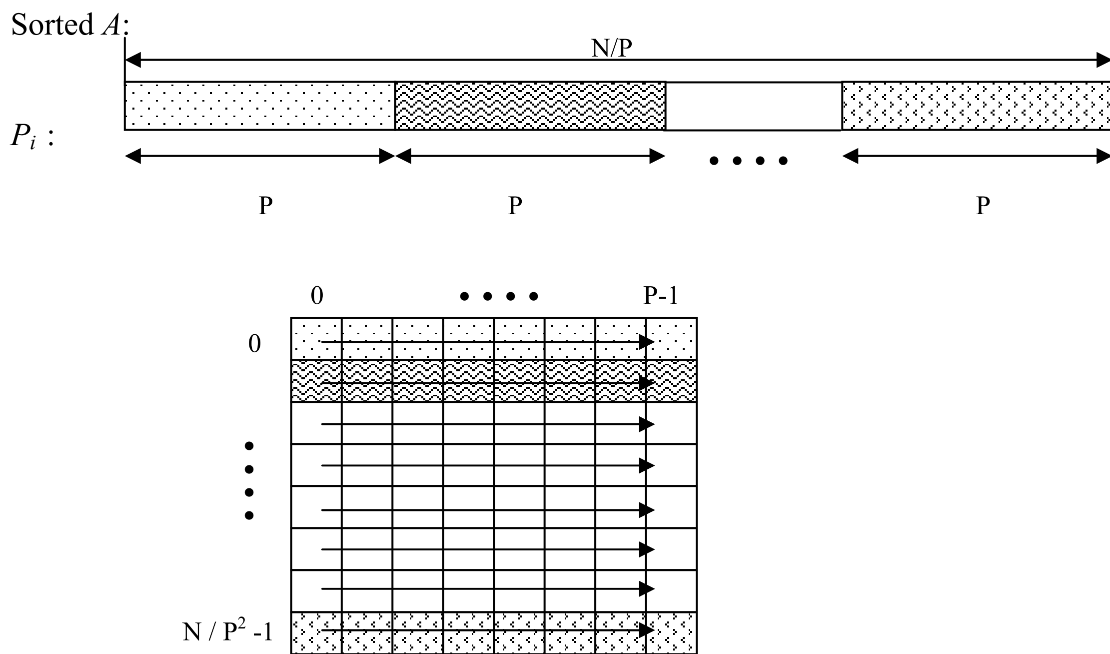


Figure 3.2 Logical representation of step 2 for one processor, arrows indicate row-major order

Step 3. Every processor reads its local 2D array column by column and generates *P* subsequences S_0, S_1, …, S_{P-1}. Such that each contains N/ P^2 numbers as shown in Figure 3.3. The resulting sequences S_0, S_1, …, S_{P-1} are sorted subsequences, since the elements within a subsequence are in the same relative order as they appeared in *A* which is already sorted after the result of the sorting algorithm in *Step 1*.

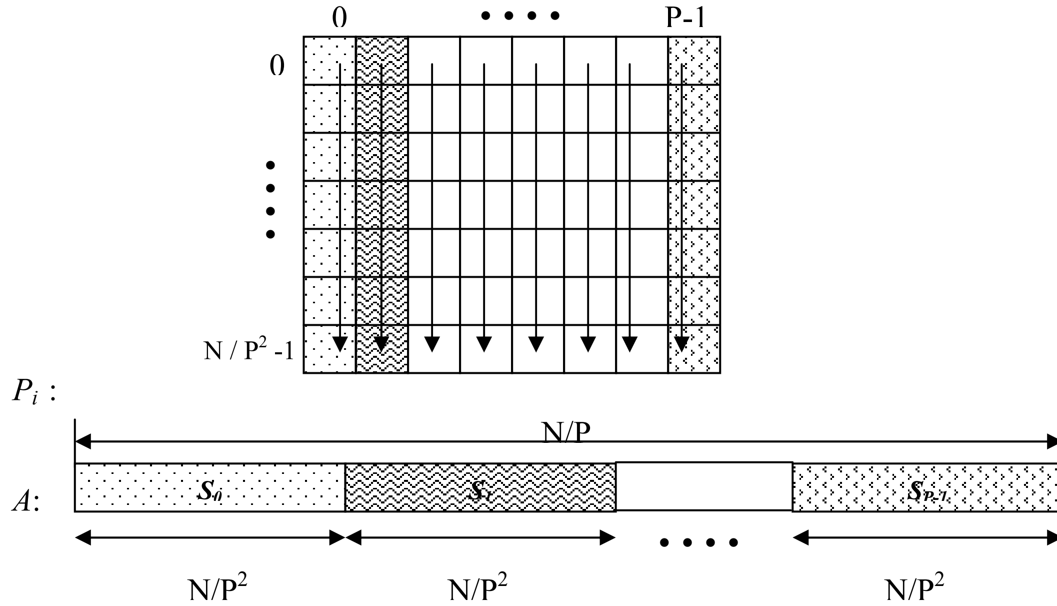


Figure 3.3 Logical representation of step 3 for one processor, arrows indicate column wise reading order

Step 4. Every processor applies the total exchange (AAPC) algorithm on A with message size N/P^2. That is the processors exchanges the generated subsequences $\boldsymbol{S_0}$, $\boldsymbol{S_1}$, …, $\boldsymbol{S_{P-1}}$ among each other in a way that $\boldsymbol{S_i}$ is send to the processor P_i. For example, processor P_0 keeps $\boldsymbol{S_0}$ itself sends $\boldsymbol{S_1}$ to processor P_1, $\boldsymbol{S_2}$ to processor P_2, and so on, while receiving $\boldsymbol{S_0}$ of P_1 from processor P_1, $\boldsymbol{S_0}$ of P_2 from processor P_2, …, $\boldsymbol{S_0}$ of P_{N-1} from processor P_{N-1}. The resulted sequence A accumulated in each processor AAPC with message size $m = N/P^2$.

Step 5. Each processor sorts A, which is resulted from the total exchange in *Step 4*, using an appropriate sequential sorting algorithm. Here, P-way merge sort, where P is the number of processors, is one of the appropriate sorting algorithms, since the p subsequences $\boldsymbol{S_0}$, $\boldsymbol{S_1}$, …, $\boldsymbol{S_{P-1}}$ are sorted subsequences. Or another alternative which I used in my implementation can be merging subsequences two by two for log$P$ steps in a binary tree fashion. This step requires O(N/P log$P$) time.

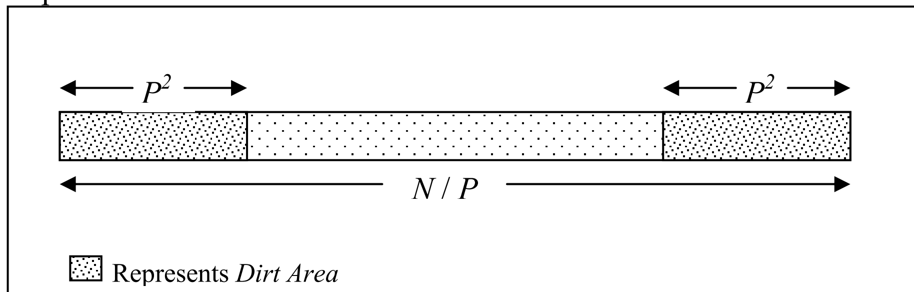
Step 6. In this step processors divide *A* into subsequences $\boldsymbol{S_0}$, $\boldsymbol{S_1}$, …, $\boldsymbol{S_{P-1}}$ of length N/P^2. That is simply $\boldsymbol{S_i}$ gets the elements a_{i*N/P^2}, $a_{(i*N/P^2)+1}$,..., $a_{i*N/P^2+N/P^2-1}$. For example, $\boldsymbol{S_0}$ gets the elements (a_0, a_1, …, a_{N/P^2-1}), $\boldsymbol{S_1}$ gets the elements (a_{N/P^2}, a_{N/P^2+1}, …, a_{2N/P^2-1}) and so on.

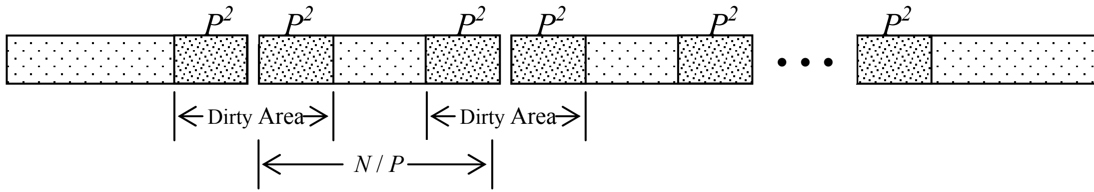
Step 7. Again *P* processors apply the total exchange (AAPC) algorithm on *A* with message size $m = N/P^2$. Therefore in this step, processors exchange their subsequences $\boldsymbol{S_0}$, $\boldsymbol{S_1}$, …, $\boldsymbol{S_{P-1}}$ among each other.

Step 8. Every processor sorts its *N/P* numbers stored in *A* using a sequential sorting algorithm parallel. As the same reasons with the *Step 5* P-way merge sort can be used here, too.

Step 9. According to Lemma 1 in [1] we know that after *Step 8*, the "*dirty area*" (i.e. unsorted portion) is P^2 length from each ends of the *A*. "*Dirty area"* is defined as the sorted subsequences within the processors whose elements may be away from their original places in the resulted sorted sequence at most P^2 distance. Since every processor sorted their elements in the Step 8, only the elements at the ends of the *A*, may not be in the correct places for the final sorted result, because they may belong to the previous or next P^2 elements. As Lemma 2 in [1] points out, if the *dirty area* falls into a place outside these P^2 elements, then the sorting in Step 8 had to be already cleared the *dirty area*. This is figured out below Figure.

For one processor





To clear the dirty area, neighboring processors exchange the first and last P^2 elements with each other while keeping their own copy, except the first and last processors. The first processor needs only to exchange its last P^2 elements, and the last needs only to exchange the first P^2 elements. Now, individual processors have all the potential elements that should reside on them in the result set. To make the algorithm more efficient, since we know the dirty areas are $2P^2$ length from ends with coming P^2 elements from the neighbors, we do not need to be sort all the array. What is more, these $2P^2$ length subsequences are combinations of two P^2 length sorted subsequences. Therefore applying merge algorithm in increasing order until obtaining the first P^2 elements (just for P^2 iterations) is enough for each processor to obtain their last P^2 elements. On the contrary case, that is for the first P^2 elements, applying the merge operation in decreasing order for P^2 iterations and getting the first P^2 elements in the reverse order will be enough to obtain correct first P^2 elements. So this step requires only $O(P^2)$ time, where P is the number of processors.

Since the input is evenly distributed among the processors and the processors are assigned by almost equal amount of work at each stage of the algorithm, **load balancing** of the multiway-merge algorithm is **evenly distributed**.

A similar approach to multiway-merge algorithm is applied for simple sorting algorithms on parallel disk systems in [17].

3.2 A Complete Example

A more sophisticated example can be obtained with $N = 256$ and $P = 4$ for observing the details in cleaning the *dirty* area, but for simplicity we used $N = 64$ numbers with $P = 4$ number of processors.

Initial array $N = 64$. Let the numbers to be sorted are:

31, 16, 3, 21, 27, 7, 6, 52, 9, 10, 11, 12, 13, 14, 15, 2, 62, 18, 40, 20, 4, 22, 23, 24, 38, 26, 5, 28, 29, 30, 1, 32, 58, 34, 35, 36, 64, 25, 59, 19, 44, 42, 43, 41, 45, 46, 47, 48, 49, 50, 51, 8, 53, 54, 55, 63, 57, 33, 39, 60, 61,17, 56, 37.

After distributing the N numbers equally to the processors, we obtain:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_0:	31	16	3	21	27	7	6	52	9	10	11	12	13	14	15	2
P_1:	62	18	40	20	4	22	23	24	38	26	5	28	29	30	1	32
P_2:	58	34	35	36	64	25	59	19	44	42	43	41	45	46	47	48
P_3:	49	50	51	8	53	54	55	63	57	33	39	60	61	17	56	37

After Step 1, sorting own data*:*

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_0:	2	3	6	7	9	10	11	12	13	14	15	16	21	27	31	52
P_1:	1	4	5	18	20	22	23	24	26	28	29	30	32	38	40	62
P_2:	19	25	34	35	36	41	42	43	44	45	46	47	48	58	59	64
P_3:	8	17	33	37	39	49	50	51	53	54	55	56	57	60	61	63

After Step 2, write data in 2D array:

P_0,

2	3	6	7
9	10	11	12
13	14	15	16
21	27	31	52

P_1,

1	4	5	18
20	22	23	24
26	28	29	30
32	38	40	62

P_2,

19	25	34	35
36	41	42	43
44	45	46	47
48	58	59	64

P_3,

8	17	33	37
39	49	50	51
53	54	55	56
57	60	61	63

After Step 3, read 2D array and generate P subsequences.

index	0	1	2	3		4	5	6	7		8	9	10	11		12	13	14	15
P_0:	2	9	13	21		3	10	14	27		6	11	15	31		7	12	16	52
P_1:	1	20	26	32		4	22	28	38		5	23	29	40		18	24	30	62
P_2:	19	36	44	48		25	41	45	58		34	42	46	59		35	43	47	64
P_3:	8	39	53	57		17	49	54	60		33	50	55	61		37	51	56	63

After Step 4, Total Exchange (AAPC)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_0:	2	9	13	21	1	20	26	32	19	36	44	48	8	39	53	57
P_1:	3	10	14	27	4	22	28	38	25	41	45	58	17	49	54	60
P_2:	6	11	15	31	5	23	29	40	34	42	46	59	33	50	55	61
P_3:	7	12	16	52	18	24	30	62	35	43	47	64	37	51	56	63

After Step 5, Sorting with successive merge operations

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_0:	1	2	8	9	13	19	20	21	26	32	36	39	44	48	53	57
P_1:	3	4	10	14	17	22	25	27	28	38	41	45	49	54	58	60
P_2:	5	6	11	15	23	29	31	33	34	40	42	46	50	55	59	61
P_3:	7	12	16	18	24	30	35	37	43	47	51	52	56	62	63	64

After Step 6, generating subsequences

index	0	1	2	3		4	5	6	7		8	9	10	11		12	13	14	15
P_0:	1	2	8	9		13	19	20	21		26	32	36	39		44	48	53	57
P_1:	3	4	10	14		17	22	25	27		28	38	41	45		49	54	58	60
P_2:	5	6	11	15		23	29	31	33		34	40	42	46		50	55	59	61
P_3:	7	12	16	18		24	30	35	37		43	47	51	52		56	62	63	64

After Step 7, Total Exchange (AAPC)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_0:	1	2	8	9	3	4	10	14	5	6	11	15	7	12	16	18
P_1:	13	19	20	21	17	22	25	27	23	29	31	33	24	30	35	37
P_2:	26	32	36	39	28	38	41	45	34	40	42	46	43	47	51	52
P_3:	44	48	53	57	49	54	58	60	50	55	59	61	56	62	63	64

After Step 8, Sorting with successive merge operations

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_0:	1	2	3	4	5	6	7	8	9	10	11	12	14	15	16	18
P_1:	13	17	19	20	21	22	23	24	25	27	29	30	31	33	35	37
P_2:	26	28	32	34	36	38	39	40	41	42	43	45	46	47	51	52
P_3:	44	48	49	50	53	54	55	56	57	58	59	60	61	62	63	64

After Step 9, Cleaning the dirty area

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_0:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P_1:	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
P_2:	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
P_3:	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

3.3 Complexity Analysis

To analyze the performance of the algorithm, following computational model from [6] is used, because it is a simple model and used in many similar works related with the work covered in this thesis. Therefore, it allows us to compare the algorithm with previous ones. In this model, powerful processors are connected by a communication network like the nodes in a complete graph. The only restrictions on the communication are the restrictions imposed by the latency and the bandwidth

properties of the network. The multiway-merge algorithm consists of a sequence of local computations interleaved with communication steps. The overall complexity of the algorithm depends on these communication and computation steps.

In order to evaluate the communication time, following cost model is used. With the assumption of no congestion, $O(\tau + \sigma m)$ will give the transfer time of a block consisting of *m* contiguous words between two processors. As stated in [6] τ is an upper bound on the latency of the network and σ is the time required for either injecting or receiving one word data from network. So $O(\tau + \sigma \max(m, p))$ will give the cost of a collective communication primitive, where m is defined as the maximum amount of data transmitted or received by a processor. This communication model is applied in several works [28], [29].

The analysis of the adapted algorithm is as follows. Define T_{comp} as the maximum time that is necessary for a processor to perform all the local computation steps. Steps 1, 2, 3, 5, 6, 8 involve no communication and are dominated by the cost of the sequential sorting in *Step 1* and the merging in *Step 5 and 8.* For the sorting in Step 1 like floating point numbers (doubles), can be sorted with merge sort and requires only $O(N/P \log(N/P))$ time. As mentioned earlier both *Step 5* and *8* (P-way merge) require $O(N/P \log P)$ time. *Steps 4 and 7* call the communication primitive AAPC with message length $m = N/P^2$. So both require $T_{comm}(N, P) \le (\tau + \sigma N(P-1)/P^2)$. And the *Step 9*, which is the final step, consists of two one-to-one communication with message length $m = P^2$, requires $T_{comm}(N, P) \le (\tau + \sigma 2P^2)$. Additionally two merge operations for input size P^2 which requires $O(P^2)$. That is also being dominated.

Hence the overall complexity of the new sorting algorithm is given by

$$T(N, P) = T_{comp}(N, P) + T_{comm}(N, P) \quad \textbf{(3.1)}$$

$$= O(N/P \log(NP) + \tau + \sigma N/P), \text{ for } N \gg P.$$

It is not exactly asymptotically optimal, but we can say that the algorithm is about asymptotically optimal with small coefficients. In general, the overall performance $T_{comp} + T_{comm}$ involves a tradeoff between T_{comp} and T_{comm} . Like many works in parallel algorithm, my aim here was also to achieve cost optimality which implies $T_{comp} = O(T_{seq} / P)$ such that T_{comm} is minimum and T_{seq} is the complexity of the best sequential algorithm. As seen from **(3.1)** , the adapted algorithm is not cost optimal. Since $P \times T(N,P)$ is not equal to T_{seq},which is $O(N\log N)$.

If we want to figure out the relative benefit of sorting the items with the adapted algorithm in parallel, then we are interested in *Speedup,* which is simply how much performance gain is achieved by the parallel algorithm over the sequential one. If we denote speedup as S, then from **(3.1)**

$$S = \frac{T_{seq}}{T(N,P)} ,$$

$$S = O\left(\frac{P\log N}{\log NP}\right) \qquad \textbf{(3.2)}$$

The efficiency of our algorithm can be calculated with using (3.2), Hence the efficiency, E is

$$\boldsymbol{E} = \frac{S}{P} = \mathbf{O}\left(\frac{\log N}{\log NP}\right) \qquad \textbf{(3.3)}$$

In the ideal case efficiency must be 1, which means all the processors devote 100 percent of their time to the computations of the algorithm. However, real parallel systems do not achieve this result, due to the communication overhead and some other factors. Generally, the difference between the total time spent by all processors

and the time required by the fastest known sequential algorithm for solving the same problem is called the *total overhead, T_o*.

$T_o = P \times T(P, N) - T_{seq}$

$T_o = O(N \log P)$ **(3.4)**

Since the T_o grows slower than T_{seq} for a fixed P, efficiency can be maintained at a desired value for increasing P, provided N is also increased. Thus the algorithm is scalable.

3.4 Design Approach

This section gives the design patterns about an efficient implementation of the multiway-merge algorithm. Multiway-merge algorithm was implemented in a high-level language so it can run on a variety of platforms. In our implementation, we used MPI (Message Passing Interface) [22, 23, 24], a specification for message passing libraries, designed to be a standard for distributed memory, message passing, parallel computing. MPI aims to provide a widely used standard for writing message-passing programs and to establish a practical, portable, efficient, and flexible standard for message passing.

Reasons for using MPI:

- *Standardization* - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms.
- *Portability* - there is no need to modify your source code when you port your application to a different platform which supports MPI.
- *Performance* - vendor implementations should be able to exploit native hardware features to optimize performance.
- *Functionality* (over 115 routines)
- *Availability* - a variety of implementations are available, both vendor and public domain.

Necessary data structure for each processor is an array of size $N/P + 2P^2$ for storing the initial numbers and the P^2 elements coming from both neighbors in Step 9. We have dedicated one processor called as master for initially creating the unsorted array length N and distributing it equally to the other processors that we called workers. Therefore, our implementation has an ***evenly distributed load balancing.*** We decided to choose this approach, in order to keep the homogeneity of the worker processors in the cluster system. By doing this, isolation of irrelevant works from the steps of the algorithm is achieved. For example, generation of the input numbers before starting the algorithm or checking the orders of the result sequences after the algorithm finishes can be done in master processor. Since these operations requires extra storage that may disturb the homogeneity.

In the first step, Quicksort could be used as a sequential algorithm if memory limitation is a concern and it gives better results in practice, otherwise merge sort theoretically gives the best time. What is more, Step 2, 3 and 6 can be achieved by reordering of the indexes of data. The merge operation in step 5 and 8 is a P-way merge since the whole sequence to be merged, consists of P sorted subsequences. In our implementation, we preferred merging subsequences two by two for $\log P$ steps in a binary tree fashion. In the last step, one possible optimization is to use the merge operation in the reverse order only for P^2 iterations to get the lowest numbers that falls to a processor. In order to find the last P^2 numbers, running the merge operation for P^2 iteration is enough.

Consequently, the algorithm requires at least $2N/P$ storage area for each processor due to the merge operation. However, some efficient inplace sequential sorting algorithm could be replaced by merge operations in step 5 and 8, if possible. Hence the sorting and merging steps could be optimized in the future works.

Chapter 4

4 Experimental Results and Analysis

4.1 Evaluating Algorithms

A number of metrics are available when evaluating a parallel algorithm for some problem. These are defined in the next few paragraphs.

4.1.1 Running Time

Since speeding up computations appears to be the reason for parallel computers, *parallel running time* is probably the most important measure in evaluating a parallel algorithm. This is defined as the time required to solve a problem, that is, the time elapsed fro the moment the algorithm starts to the moment it terminates. Running time is usually obtained by counting two kinds of steps executed by the algorithm: routing steps and computational steps. In a routing step, data travel from one processor to another through the communication network or via the shared memory. A computational step, on the other hand, is an arithmetic or logic operation performed on data within a processor. For a problem of size n, the parallel worst-case running time of an algorithm, a function of n, will be denoted by $t(n)$.

A good indication of the quality of a parallel algorithm for some problem is the *speedup* it produces. This is defined as

$$\text{Speedup} = \frac{\text{Worst-case running time of fastest known sequential algorithm for the problem}}{\text{Worst-case running time of parallel algorithm}} \qquad \textbf{(4.1)}$$

It is clear that the larger the ratio, the better the parallel algorithm. Ideally, of course, one hopes to achieve a speedup of N when solving a problem using N processors operating in parallel. In practice, such a speed up cannot generally be achieved since

(1) in most cases it is impossible to decompose a problem into N tasks each requiring $1/N$ of the time taken by one processor to solve the original problem, and

(2) the structure of the parallel computer used to solve a problem usually imposes restrictions that render the desired running time unattainable.

4.1.2 Number of Processors

Another criterion for assessing the value of a parallel algorithm is the *number of processors* it requires to solve a problem. Clearly, the larger the number of processors, the more expensive the solution becomes to obtain. For a problem of size n, the number of processors required by an algorithm, a function of n, will be denoted by $p(n)$. the processors, numbered 1 to $p(n)$, will be denoted by $P_1, P_2, \ldots, P_{p(n)}$,.

4.1.3 Cost

The *cost* of a parallel algorithm is defined as the product of the previous two measures; hence

Cost = parallel running time x number of processors used. **(4.2)**

In other words, cost equals the number of steps executed in solving a problem in the worst case. If a lower bound is known on the number of sequential operations required in the worst case to solve a problem and the cost of a parallel algorithm for the problem matches this lower bound to within a constant multiplicative factor, the algorithm is said to be *cost-optimal,* since any parallel algorithm can be simulated on a sequential computer. In the particular case of sorting, a parallel algorithm whose cost is $O(n \log n)$ will be cost-optimal. Alternatively, when a lower bound is not known, the *efficiency* of the parallel algorithm, defined as

$$\text{Efficiency} = \frac{\text{Worst-case running time of fastest known sequential algorithm for the problem}}{\text{Cost of parallel algorithm}} \quad \textbf{(4.3)}$$

Is used to evaluate its cost. In the most cases,

$$\text{Efficiency} \leq 1;$$

Otherwise a faster sequential algorithm can be obtained from the parallel one!
For a problem of size n, the cost of a parallel algorithm, a function of n, will be denoted by $c(n)$. Thus $c(n) = t(n) \times p(n)$.

4.1.4 Other Measures

Besides the three criteria outlined above, other measures are sometimes used to evaluate parallel algorithms. Like the chip area, length of the communication wires, period of a circuit are important for the VLSI technology, which is used in most parallel computers. However these measures are not necessary for the work presented here.

4.2 Experimental Data

The algorithm was implemented using MPI and run on *BORG* system, which is a 32 node PC-cluster. Our experimental results illustrate the efficiency and the scalability of our algorithm. The results are competitive with previous works.

For each experiment, the input is evenly distributed amongst the processors. The output consists of the elements in non-descending order arranged amongst the processors so that the elements at each processor are in sorted order and no element at processor P_i is greater than any element at processor P_j, for all $i < j$. In other words, the final sorted data array remains distributed among the different processors at the end of the sort.

Quicksort was used to sequentially sort integers. Wherever possible, we tried to use the MPI standards in our implementations, to prevent vendor-specific properties. In fact, MPI does provide all of our communication primitives as part of its collective Communication Library. Since MPI becomes the universal standard in Parallel Computing gradually, we have preferred it, so we could able to compare the algorithm with similar ones also implemented in MPI.

We tested our code with 4-byte *integer* numbers on three different conventional benchmarks. Firstly, for generating the input, we have applied the ***Uniform benchmark [U]***, which requires a uniformly distributed random input, obtained by calling the C library random number generator *random().* The values are uniformly distributed in the range 0 through $2^{32} – 1$. Secondly, we tested our code on a ***Gaussian [G]*** distributed input, which is obtained by calling the *random()* function four times and then taking the average value for each input key. Finally, we have used an input that consists of all zero values, ***Zero [Z]***. In the previous works related on this topic, these benchmarks are accepted as standard. [7]

In order to preserve the homogeneity of the nodes (worker processors), we have isolated the heterogeneous code from the core one which is executed similarly in all worker processors. As mentioned in implementation details section, one node called master is dedicated for generation of the unsorted sequence and distribution of it evenly to the worker processors. To generate the unsorted sequence in *Uniform Benchmark [U]*, we have applied the following strategy. Firstly, the master node generates *N* numbers starting from 0 to *N*-1. Next it permutes the array, calling *random()* function repeatedly to shuffle the array. After execution, the master node also gathers the resulted sequences from each worker, and checks whether the total result was correctly sorted or not.

We have measured the time by calling the MPI library function *MPI_Wtime ()* first at the start of the algorithm and next at the end of the step 9, so the difference will give the execution time of the algorithm. As is the convention in the literature, the data to be sorted is already distributed among the processors before the timing is begun. Since each processor could have different execution times, we have put *MPI_Barrier ()* before each *MPI_Wtime ()* to keep the synchronicity. Thus the time measured is the maximum execution time of the individual execution times of worker processors. During the experiments, no other MPI programs or processes that require the processors of *BORG* were allowed to execute.

The execution times for various input sizes with different processors are given in Table 4.1 for *Uniform benchmark,* Table 4.2 for *Gaussian benchmark*, and Table 4.3 for *Zero benchmark*. Each data point reported in the tables below represent the average results over five different sets of random data. More detailed versions of these tables could be found in Appendices. Each set of data is created using a different random number generator seed value. The performance of multiway-merge parallel sorting algorithm as a function of input size is represented by graphics in Figure 4.2 .

Sequential versus Parallel Run time in seconds						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1,335011	2,820501	5,965523	12,45525	24,03498	50,40286
2	0,839626	1,739909	3,62222	7,554462	15,8012	30,75736
4	0,484416	1,036845	2,085567	4,258955	8,859255	18,02189
8	0,273475	0,590014	1,204814	2,418979	4,805135	9,790042
16	0,154148	0,326091	0,813763	1,832427	2,911867	5,367821

Table 4.1 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [U] with PSORT on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input.

Sequential versus Parallel Run time in seconds						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1.311262	2.804137	5.85161	12.29549	23.93699	50.39432
2	0.824045	1.720443	3.589029	7.536904	15.62571	30.52727
4	0.482399	1.008216	2.078269	4.264622	8.783141	17.95082
8	0.274923	0.576005	1.31283	2.490565	4.842571	9.800294
16	0.144154	0.31275	1.058176	1.81683	2.885767	5.281401

Table 4.2 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [G] with PSORT on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input.

Sequential versus Parallel Run time in seconds						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	0.609718	1.27802	2.675383	5.560864	17.33328	35.99081
2	0.477112	0.985661	2.060151	4.184864	8.532312	23.82078
4	0.308333	0.642771	1.305432	2.695743	5.364357	10.5972
8	0.193543	0.395494	0.943082	1.615056	3.115139	6.142575
16	0.121054	0.2328	0.880503	1.527776	2.07019	3.588378

Table 4.3 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [Z] on with PSORT various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input.

As expected, the run time is cut by almost a factor of two when increasing number of processors to twice as the previous. But it is not exactly two, because of the communication and interprocessor computations. The execution time is nearly inversely with the number of processors. However the ratio between execution times is decreasing when the number of processors increases. That is the gain obtained by using 16 processors instead of 8 is smaller than the gain obtained by using 8 processors instead of 4. As the number of processors increases, each processor performs less computation but the amount of communication is not decreased with same ratio. In other words, the communication time is comparable to the computation time, because there are always just constant number of physical wires that connects the nodes of the cluster to the network and the local network’s bandwidth remains constant. When the problem size and the number of processors increase, more communication overhead is added.

As seen from the results (Figure 4.1), the performance of the algorithm is not dependent tightly on the input distribution. The general characteristics of the performance function are observed for all three benchmarks similarly. The execution time for [Z] is smaller than the others, which is what we expect.

Also some deviations are observed for specific input sizes. We expect the reason for them is due to the *all to all personalized communication* in the implementation. When we have analyzed the distribution of the time for the algorithm in Figure 4.8 and Figure 4.9, we have discovered that the standard MPI communication primitive for *all to all personalized communication* is not as efficient as expected. MPI keeps some internal buffering mechanism to prevent the congestion, thus the cache size to hold that buffer plays an important role. The extra synchronization affects the performance in a negative way.

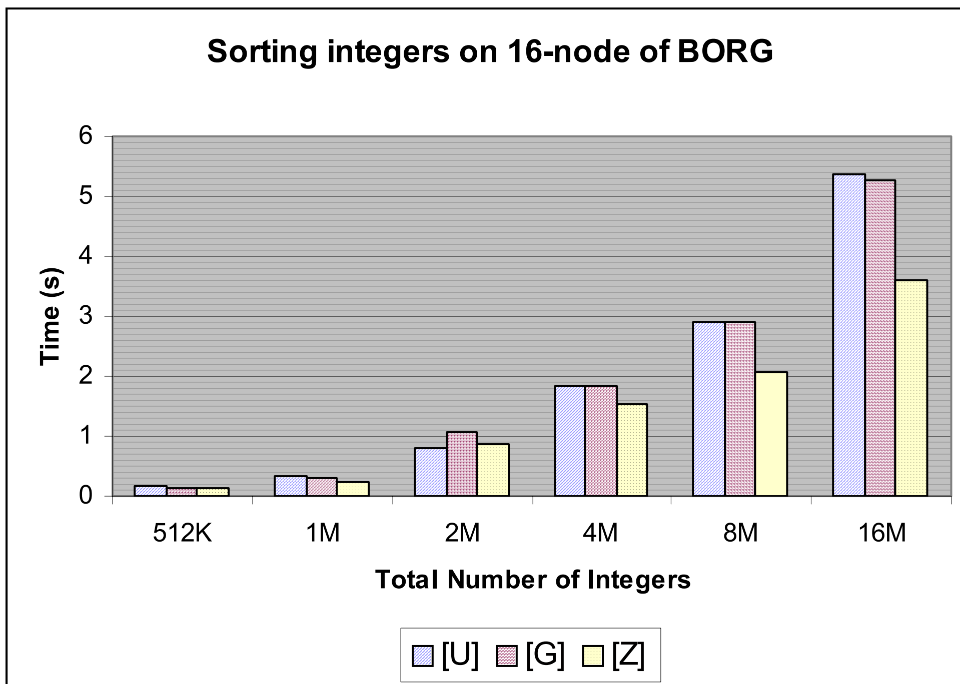


Figure 4.1 Comparison of different benchmarks, while sorting on 16 nodes of *BORG with PSORT*

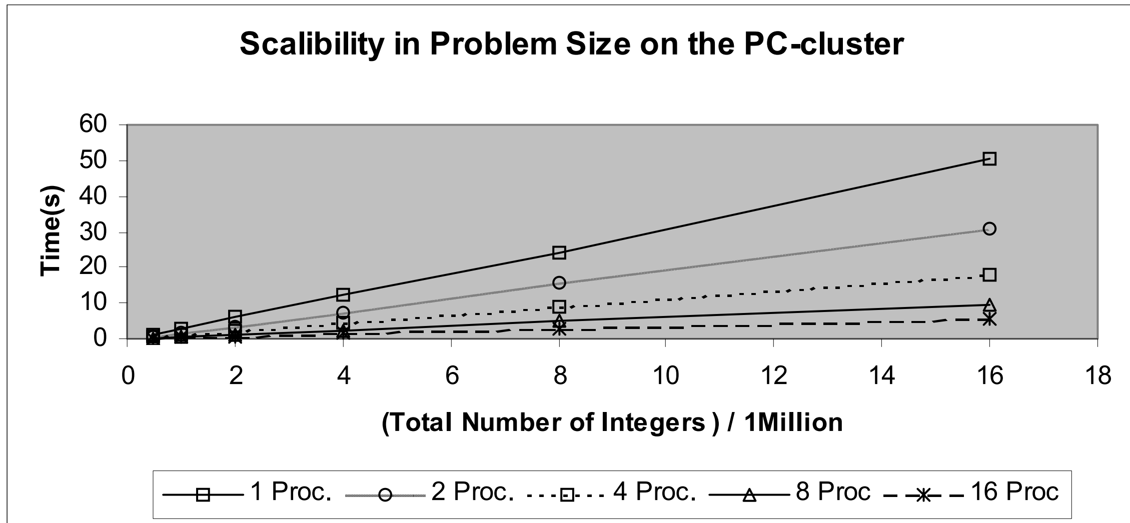


Figure 4.2 Scalibility of sorting *integers* [U] with respect to problem size, for different numbers of processors using PSORT.

We can examine the scalability of multiway-merge sorting as a function of problem size for differing number of processors. As seen in Figure 4.2, there exists an almost linear dependence between the execution time and the input size for a fixed number of processors. This relationship is also illustrated in Figure 4.3 obviously for a fixed number (16) of processors. On the other way, to view the scalability in another perspective, Figure 4.4 marks the change between execution times for different number of processors, while sorting 8M integer data. If the number of processors increases, the graphic approximates to a stable level, which means after some point increase in the number of processors won't affect the performance too much.

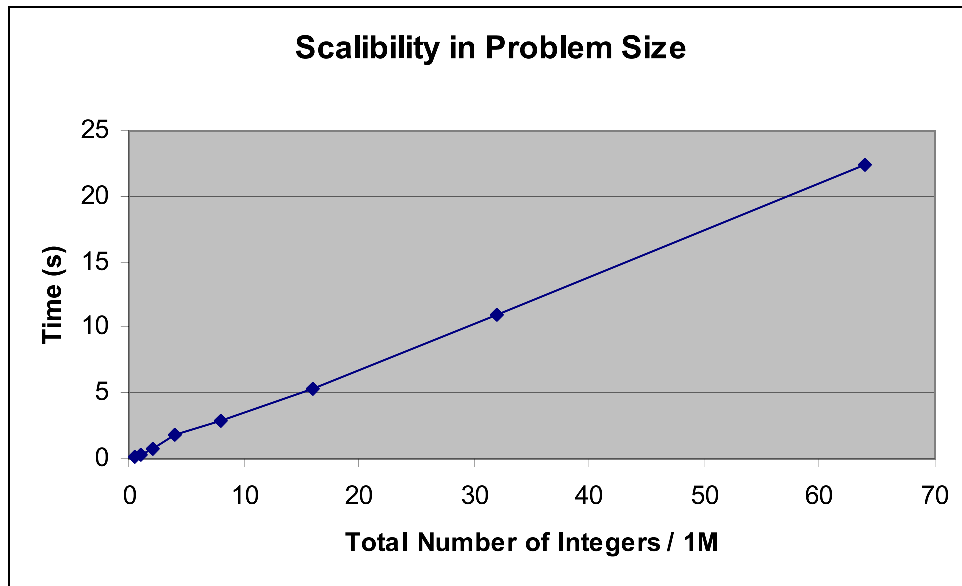


Figure 4.3 Scalability in problem size for 16 nodes

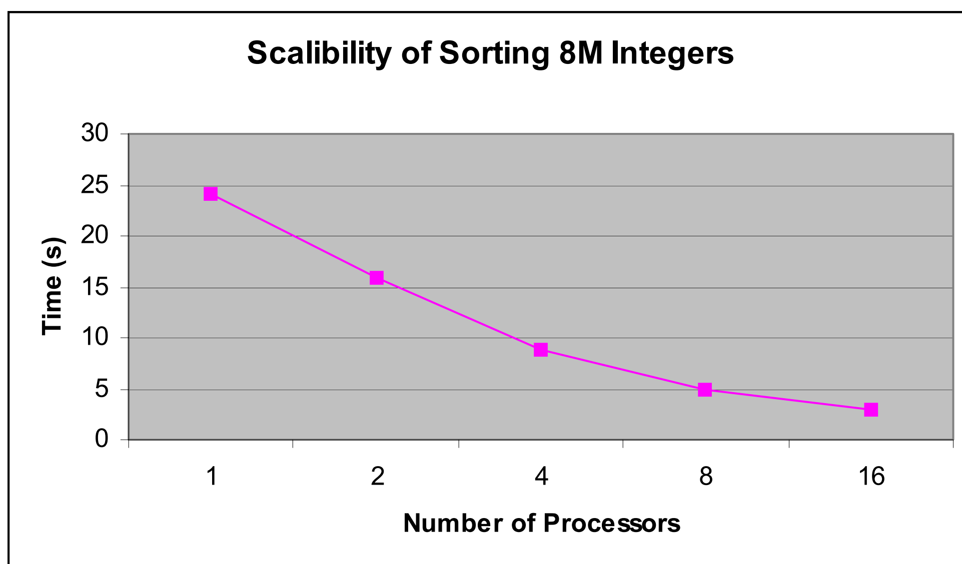


Figure 4.4 Scalibility of sorting 8M *integers* for different number of processors

Speedup = Tsequential / Tparallel						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1	1	1	1	1	1
2	1,590008	1,621063	1,646924	1,648727	1,521086	1,638725
4	2,755919	2,720272	2,860384	2,924484	2,71298	2,796758
8	4,881661	4,780395	4,951407	5,148969	5,001937	5,14838
16	8,660593	8,64943	7,330785	6,79713	8,25415	9,389816

Table 4.4 Speedup values of PSORT for different input [U] sizes on various number of processors

Speedup = Tsequential / Tparallel						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1	1	1	1	1	1
2	1.59125	1.629892	1.630416	1.631372	1.531897	1.650797
4	2.718212	2.781285	2.815618	2.883138	2.725334	2.807355
8	4.769564	4.868251	4.457247	4.93683	4.943034	5.142123
16	9.096283	8.966059	5.5299	6.767554	8.294846	9.541846

Table 4.5 Speedup values of PSORT for different input [G] sizes on various number of processors

Speedup = Tsequential / Tparallel						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1	1	1	1	1	1
2	1.277935	1.296612	1.298635	1.328804	2.031486	1.5109
4	1.977465	1.988298	2.049424	2.062831	3.231194	3.396257
8	3.150292	3.231454	2.83685	3.443141	5.564206	5.859239
16	5.036732	5.489767	3.038472	3.639843	8.372796	10.02983

Table 4.6 Speedup values of PSORT for different input [Z] sizes on various number of processors

Another way of viewing the same data is to use the so called fixed speedup, computed as the ratio of the time it takes to run a problem on a processor to the time is take to run the same problem on a given number of processors. Figure 4.5, Figure 4.6, and Figure 4.7 show the fixed speedup versus the number of processors on different sizes of input. For a fixed number of processors, increasing the problem size increases the speedup efficiency. Actually, the speedup curve grows closer to linear as the problem size is incrementally increased. As the number of processors increases, the fixed speedup curve deviates from a straight line and starts to saturate. It would be very crucial to expect exactly O(*N/P*) time as processors are added. For a fixed problem size, adding more processors to the solution results in diminishing speedup returns. The speedup results are respectable for *BORG,* a cluster based distributed memory MIMD architecture. They might be improved with a more optimized implementation. It is important to keep in mind the fact that communication over a LAN between cluster nodes is expensive.

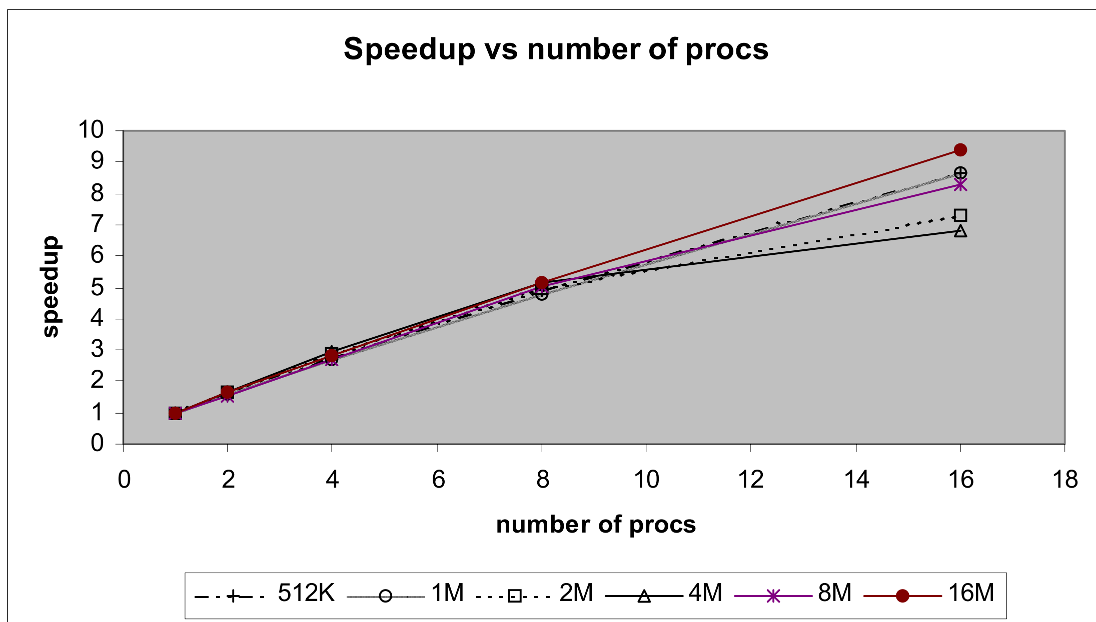


Figure 4.5 Speedup versus number of processors for different sizes of input [U]

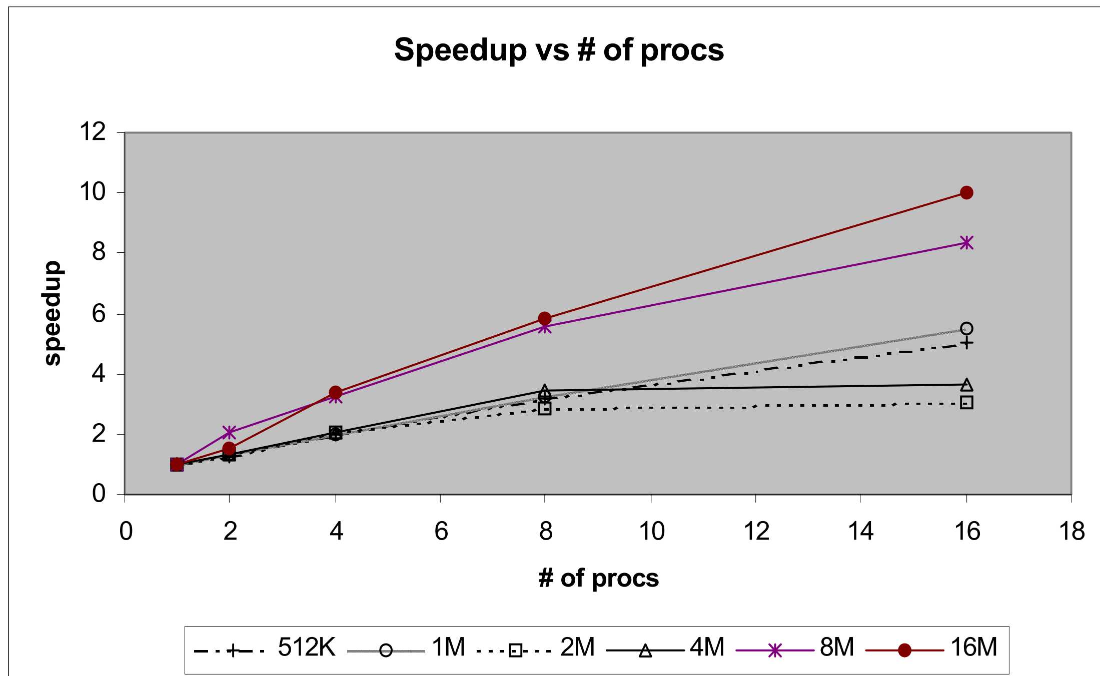


Figure 4.6 Speedup versus number of processors for different sizes of input [G]

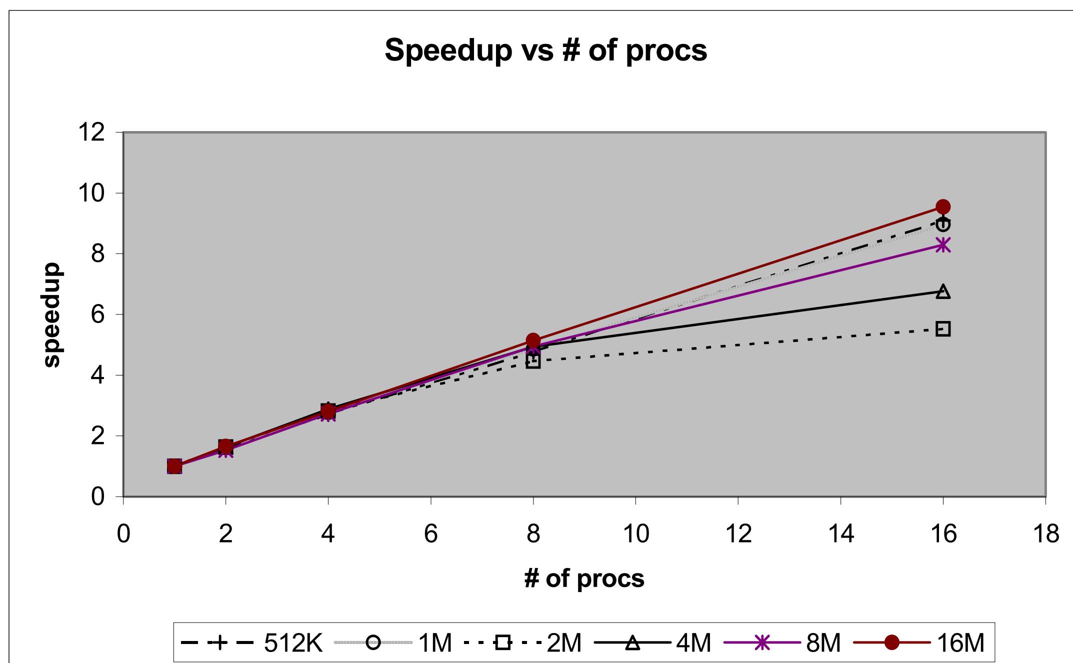


Figure 4.7 Speedup versus number of processors for different sizes of input [Z]

As observed in three of the speedup figures, the speedup for the input data of size 2M and 4M is smaller than the others. This is because the small size inputs like 512K and 1M are small enough to fit the interior cache size for buffer. On the other hand, for the large sizes of inputs like 8M and 16M, the ratio of computation over communication is large enough to suppress the loss comes from buffering. Hence, the actual behavior of the algorithm is observed for large *N* over *P*. It is clear that, larger data sets more effectively offset the overheads of the algorithm.

efficiency = speedup / # of processors						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1	1	1	1	1	1
2	0,795004	0,810531	0,823462	0,824364	0,760543	0,819363
4	0,68898	0,680068	0,715096	0,731121	0,678245	0,699189
8	0,610208	0,597549	0,618926	0,643621	0,625242	0,643547
16	0,541287	0,540589	0,458174	0,424821	0,515884	0,586864

Table 4.7 Efficiency of PSORT for various numbers of processors on different input [U] sizes

efficiency = speedup / # of processors						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1	1	1	1	1	1
2	0.795625	0.814946	0.815208	0.815686	0.765949	0.825399
4	0.679553	0.695321	0.703904	0.720785	0.681333	0.701839
8	0.596196	0.608531	0.557156	0.617104	0.617879	0.642765
16	0.568518	0.560379	0.345619	0.422972	0.518428	0.596365

Table 4.8 Efficiency of PSORT for various numbers of processors on different input [G] sizes

efficiency = speedup / # of processors						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1	1	1	1	1	1
2	0.638967	0.648306	0.649317	0.664402	1.015743	0.75545
4	0.494366	0.497075	0.512356	0.515708	0.807798	0.849064
8	0.393787	0.403932	0.354606	0.430393	0.695526	0.732405
16	0.314796	0.34311	0.189904	0.22749	0.5233	0.626864

Table 4.9 Efficiency of PSORT for various numbers of processors on different input [Z] sizes

The Table 4.7 gives the efficiency of multiway-merge sorting algorithm on various number of processors for different input sizes. Efficiency is a measure of how the observed speedup compares with linear speedup, which is ideal. In other words, efficiency measures the portion of the time each processor spends doing "useful" work that contributes to the final solution. As seen from the values, the efficiency of our implementation is not super, but it is not going to be unreasonably bad either. When the execution times spent for each step were examined in Figure 4.8 and Figure 4.9 it can be seen that most of the time has spent to sequential sort in Step1. This step can be optimized using better sequential sorting algorithms for specific keys to be sorted. For example using *Radix Sort* for integers will give O(*N*). The next biggest time consuming steps are the communication steps that require AAPC for transpose operation. Although, we did not spent special effort to implement the AAPC, with an optimized version the performance could be improved. There are special interest groups like [35] which are professionally working on the *MPI_AlltoAll()* to making an improved version nowadays. Additionally, there are many recent works in literature about efficient data distribution between processors. There are efficient algorithms for block-cyclic array redistribution between processors like in [21].

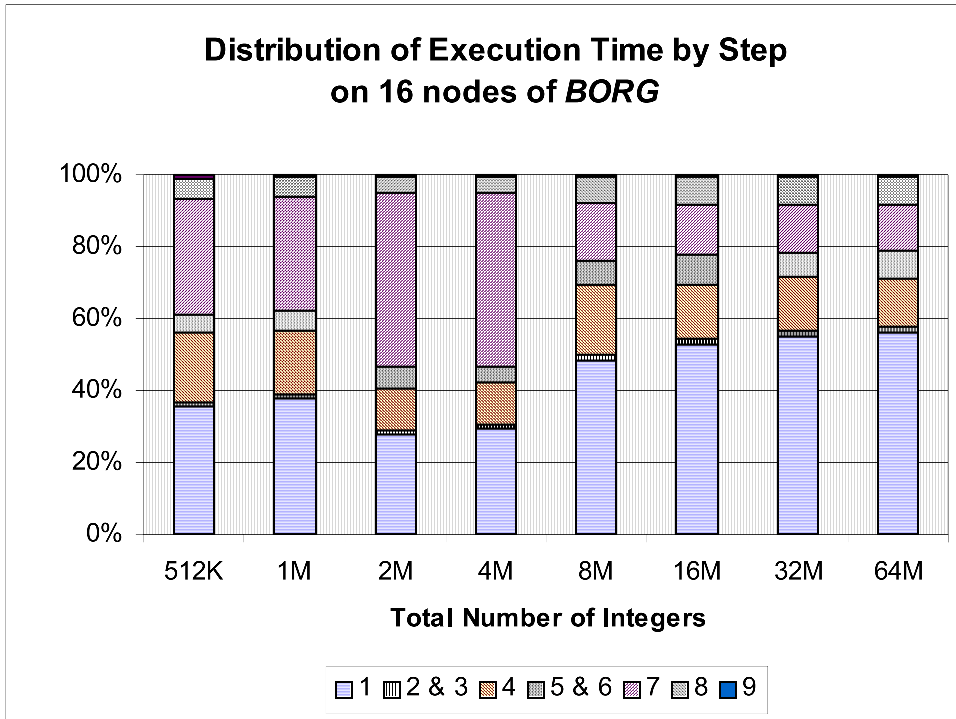


Figure 4.8 Distribution of execution time by step on 16 nodes of *BORG*

Figure 4.9 Distribution of execution time by step on 8 nodes of *BORG*

4.3 Comparisons of the Results

There are many factors that could quickly speeds up the execution times of the experiments. For example, using vendor-specific communication primitives or machine specific implementations can boost up the results. What is more, there are also implementation specific factors like using fast accessible registers for frequently used variables, or using faster methods for memory access, copy, or comparison operations etc. and some other tricks might also affect the results. In our implementation, we have applied some of these. However, our goal here is to investigate the potential performance of the multiway-merge algorithm in general, and not to benchmark specific machines or implementations. Therefore, our implementation is not optimized to the hardware. Although it performs comparable results, the implementation can be optimized more.

As it is to be emphasized in [7], although there are many theoretical interests in parallel sorting, we are able to find relatively few empirical studies. Most of the empirical studies that we found were obtained from tests on old-fashioned machines or for special designed parallel computers, not today's pc-clusters. Additionally, some of the works that we have found did not supply convenient data for comparison like speedup. This work is also important due to its property of being the some of the unique studies that supply empirical results for parallel sorting on a PC cluster technology.

In order to compare our results with other algorithms, we have decided to implement two more algorithms- one from parallel sample sort algorithms and one from quicksort algorithms with MPI on our *BORG* cluster. Otherwise the comparison results may not be adequate due to the software and hardware dependent factors like programming, network speed, power of processors, and memory etc. We have implemented PSRS (Parallel Sorting by Regular Sampling) as an instance of sample sort algorithms – also known single step algorithms as mentioned in chapter 2-, and

hypercube quicksort algorithm as an instance for multistep algorithms. Both algorithms are efficient algorithms in their classes. We have run the three algorithms in the same cluster and obtained the experimental results below. For each algorithm we tried the three benchmarks U uniformly distributed, G Gaussian distributed, and Z zero filled input data. The experimental results for Hyperquicksort are as follows:

Sequential versus Parallel Run time in seconds						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1,335011	2,820501	5,965523	12,45525	24,03498	50,40286
2	0,656181	1,36175	2,846882	5,733281	11,86855	28,24622
4	0,422815	0,877076	1,800701	3,737121	7,74239	15,1991
8	0,26118	0,540686	1,118924	2,242671	4,573165	9,146568
16	0,170847	0,32431	0,648274	1,339215	2,64014	5,456549

Table 4.10 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M integers [U] with Hyperquicksort on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential Quicksort algorithm on the input.

Sequential versus Parallel Run time in seconds						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1,311262	2,804137	5,85161	12,29549	23,93699	50,39432
2	0,647346	1,342474	2,689541	5,763189	11,60979	26,71161
4	0,407888	0,854147	1,776614	3,688194	7,573267	15,93311
8	0,262176	0,539236	1,132204	2,256625	4,864059	9,718607
16	0,159005	0,325405	0,656986	1,343272	2,812576	5,660201

Table 4.11 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M integers [G] with **Hyperquicksort** on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential Quicksort algorithm on the input.

Sequential versus Parallel Run time in seconds						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	0,609718	1,27802	2,675383	5,560864	17,33328	35,99081
2	0,546109	1,11942	2,302662	4,70867	9,660501	21,80971
4	0,356734	0,726149	1,480463	3,025802	6,194669	12,71959
8	0,224758	0,45141	0,971524	1,858256	3,772465	7,656362
16	0,133912	0,270888	0,556594	1,108306	2,267947	4,507517

Table 4.12 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M integers [Z] with **Hyperquicksort** on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential Quicksort algorithm on the input.

For the PSRS, [Z] distribution is not applicable since the algorithm based on the regular sampling heuristic. When all elements are zero the pivot values could not able to partition the local samples so it might work for some cases worst than the sequential algorithm due to the extra communication overhead. The execution times for [U] and [G] are given below:

Sequential versus Parallel Run time in seconds						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1,335011	2,820501	5,965523	12,45525	24,03498	50,40286
2	0,767775	1,631442	3,352483	7,085041	14,69197	28,69112
4	0,406358	0,855627	1,755299	3,739375	7,702234	16,07232
8	0,220271	0,469623	0,966984	2,100859	4,203759	8,432961
16	0,143817	0,252677	0,526959	1,146468	2,46587	4,766118

Table 4.13 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M integers [U] with **PSRS** on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential Quicksort algorithm on the input.

Sequential versus Parallel Run time in seconds						
# of procs	input size					
	512K	1M	2M	4M	8M	16M
1	1,311262	2,804137	5,85161	12,29549	23,93699	50,39432
2	0,764684	1,599278	3,411533	7,056921	14,70411	28,57786
4	0,418736	0,890938	1,945153	3,866342	7,984475	16,73927
8	0,244732	0,501521	1,181397	2,166164	4,461673	9,241586
16	0,135747	0,276674	0,610951	1,197954	2,441376	4,913464

Table 4.14 Total execution time (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [G] with **PSRS** on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input.

For the sake of simplicity in the next following paragraphs we will present the related data together for these three algorithms. In order to eliminate duplication of the data, we decided to put the detailed experimental results in the Appendices. We present the experimental results in the tables which we call *comparison charts,* in a three dimensional view- one for each type of algorithm, one for the number of inputs, and one for the number of processors. Whenever it is possible, we create a comparison chart for each different input distributions like [U], [G], and [Z]. To make the sizes of comparison charts more compact, the experimental data results are truncated up to the second digit after comma, however detailed results are also supported in appendices. *PSORT* refers *Multiway-merge Parallel Sorting Algorithm*, *PSRS* refers *Parallel Sorting with Regular Sampling* and QSORT refers *Hyperquicksort*. The following tables are showing the execution times of the algorithms for various number of inputs and processors, corresponding speedup values and efficiencies respectively. The values of [19] are presented under the column of PSRS (Parallel Sorting with regular Sampling), our results are presented under the column PSORT, where it stands for Multiway-merge Parallel Sorting Algorithm and the results for hyperquicksort algorithm is presented under the column QSORT. The results are discussed after the tables.

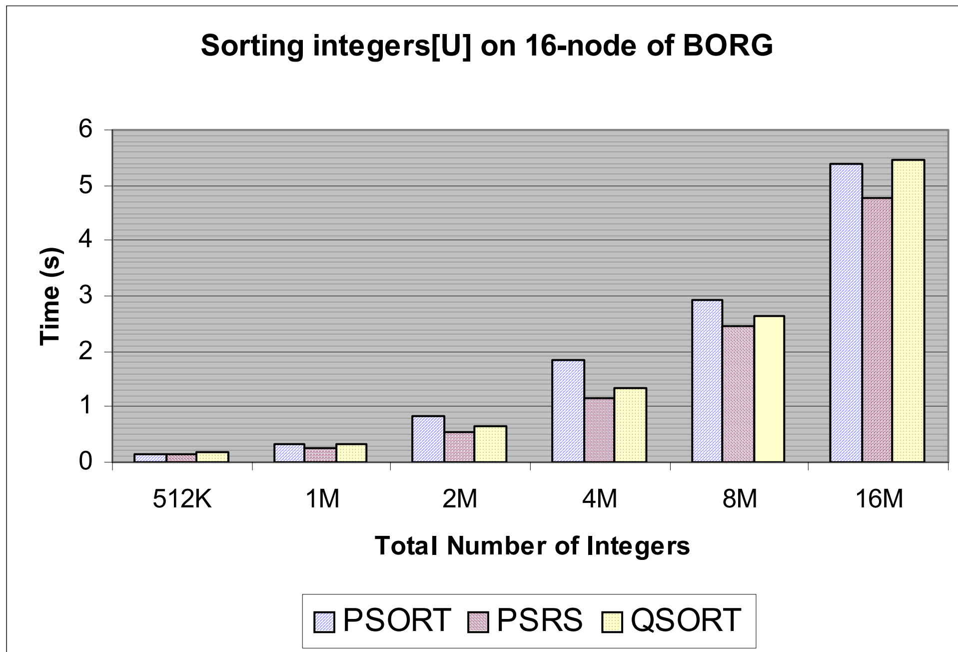


Figure 4.10 Execution time versus number of uniformly distributed [U] inputs

Comparison Chart [U]																		
Sequential versus Parallel Run times in seconds																		
	input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
1	1,34	1,34	1,34	2,82	2,82	2,82	5,97	5,97	5,97	12,46	12,46	12,46	24,03	24,03	24,03	50,40	50,40	50,40
2	0,84	0,77	0,66	1,74	1,63	1,36	3,62	3,35	2,85	7,55	7,09	5,73	15,80	14,69	11,87	30,76	28,69	28,25
4	0,48	0,41	0,42	1,04	0,86	0,88	2,09	1,76	1,80	4,26	3,74	3,74	8,86	7,70	7,74	18,02	16,07	15,20
8	0,27	0,22	0,26	0,59	0,47	0,54	1,20	0,97	1,12	2,42	2,10	2,24	4,81	4,20	4,57	9,79	8,43	9,15
16	0,15	0,14	0,17	0,33	0,25	0,32	0,81	0,53	0,65	1,83	1,15	1,34	2,91	2,47	2,64	5,37	4,77	5,46

Table 4.15 Comparison of total execution times (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [U] with **PSORT, PSRS, and QSORT** on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input.

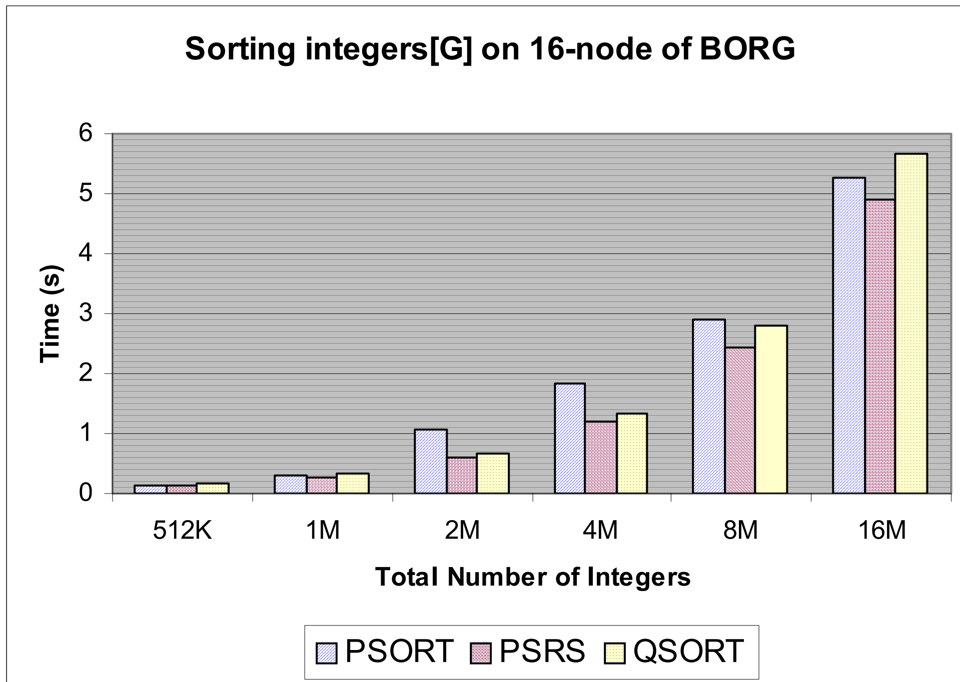


Figure 4.11 Execution time versus number of Gaussian distributed [G] inputs

Comparison Chart [G]																		
Sequential versus Parallel Run times in seconds																		
	input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
1	1,31	1,31	1,31	2,80	2,80	2,80	5,85	5,85	5,85	12,30	12,30	12,30	23,94	23,94	23,94	50,39	50,39	50,39
2	0,82	0,76	0,65	1,72	1,60	1,34	3,59	3,41	2,69	7,54	7,06	5,76	15,63	14,70	11,61	30,53	28,58	26,71
4	0,48	0,42	0,41	1,01	0,89	0,85	2,08	1,95	1,78	4,26	3,87	3,69	8,78	7,98	7,57	17,95	16,74	15,93
8	0,27	0,24	0,26	0,58	0,50	0,54	1,31	1,18	1,13	2,49	2,17	2,26	4,84	4,46	4,86	9,80	9,24	9,72
16	0,14	0,14	0,16	0,31	0,28	0,33	1,06	0,61	0,66	1,82	1,20	1,34	2,89	2,44	2,81	5,28	4,91	5,66

Table 4.16 Comparison of total execution times (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [G] with **PSORT, PSRS, and QSORT** on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input.

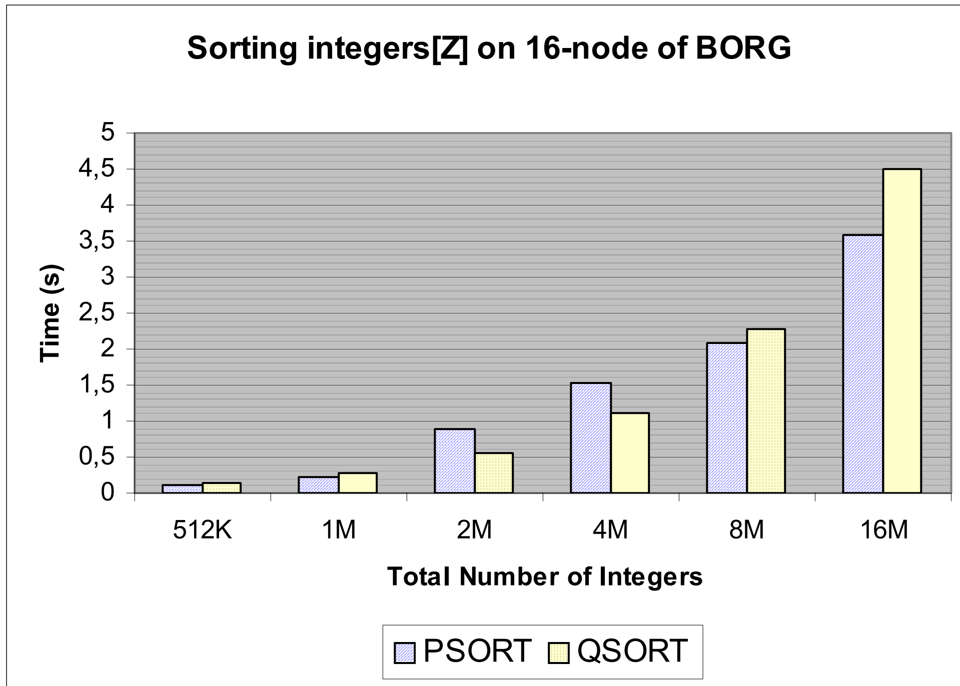


Figure 4.12 Execution time versus number of zero distributed [Z] inputs

Comparison Chart [Z]																		
Sequential versus Parallel Run times in seconds																		
	input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
1	0,61	0,61	0,61	1,28	1,28	1,28	2,68	2,68	2,68	5,56	5,56	5,56	17,33	17,33	17,33	35,99	35,99	35,99
2	0,48	NA	0,55	0,99	NA	1,12	2,06	NA	2,30	4,18	NA	4,71	8,53	NA	9,66	23,82	NA	21,81
4	0,31	NA	0,36	0,64	NA	0,73	1,31	NA	1,48	2,70	NA	3,03	5,36	NA	6,19	10,60	NA	12,72
8	0,19	NA	0,22	0,40	NA	0,45	0,94	NA	0,97	1,62	NA	1,86	3,12	NA	3,77	6,14	NA	7,66
16	0,12	NA	0,13	0,23	NA	0,27	0,88	NA	0,56	1,53	NA	1,11	2,07	NA	2,27	3,59	NA	4,51

Table 4.17 Comparison of total execution times (in seconds) for sorting 512K, 1M, 2M, 4M, 8M, 16M *integers* [Z] with **PSORT, PSRS, and QSORT** on various processors in the BORG PC cluster system. One processor data is obtained by running a sequential *Quicksort* algortihm on the input. NA means *Not Applicable*

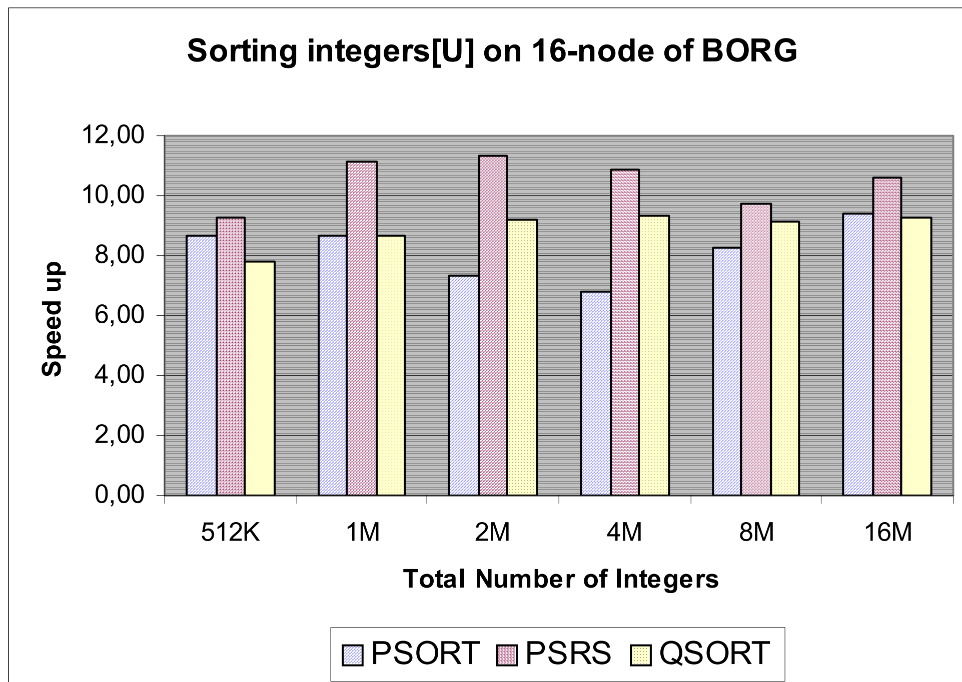


Figure 4.13 Speed up versus number of uniformly distributed [U] inputs

Comparison Chart [U]																		
Speedup = Tsequential / Tparallel																		
	input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
2	1,59	1,74	2,03	1,62	1,73	2,07	1,65	1,78	2,10	1,65	1,76	2,17	1,52	1,64	2,03	1,64	1,76	1,78
4	2,76	3,29	3,16	2,72	3,30	3,22	2,86	3,40	3,31	2,92	3,33	3,33	2,71	3,12	3,10	2,80	3,14	3,32
8	4,88	6,06	5,11	4,78	6,01	5,22	4,95	6,17	5,33	5,15	5,93	5,55	5,00	5,72	5,26	5,15	5,98	5,51
16	8,66	9,28	7,81	8,65	11,16	8,70	7,33	11,32	9,20	6,80	10,86	9,30	8,25	9,75	9,10	9,39	10,58	9,24

Table 4.18 Comparison of speedup values for different input [U] sizes on various number of processors for PSORT, PSRS, QSORT algorithms

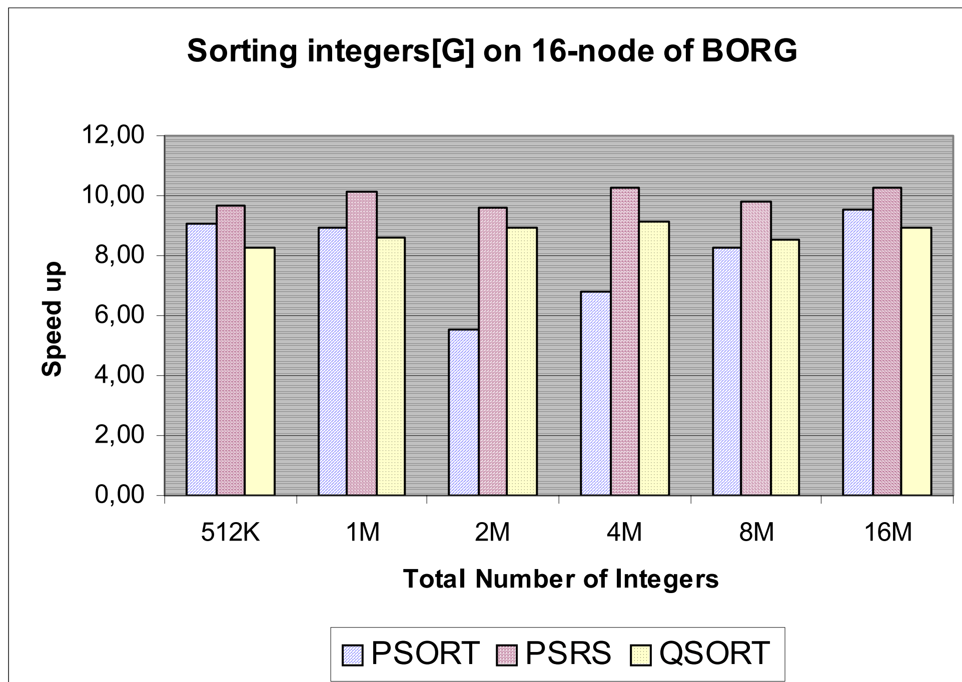


Figure 4.14 Speed up versus number of Gaussian distributed [G] inputs

Comparison Chart [G]																		
Speedup = Tsequential / Tparallel																		
	Input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
2	1,59	1,71	2,03	1,63	1,75	2,09	1,63	1,72	2,18	1,63	1,74	2,13	1,53	1,63	2,06	1,65	1,76	1,89
4	2,72	3,13	3,21	2,78	3,15	3,28	2,82	3,01	3,29	2,88	3,18	3,33	2,73	3,00	3,16	2,81	3,01	3,16
8	4,77	5,36	5,00	4,87	5,59	5,20	4,46	4,95	5,17	4,94	5,68	5,45	4,94	5,37	4,92	5,14	5,45	5,19
16	9,10	9,66	8,25	8,97	10,14	8,62	5,53	9,58	8,91	6,77	10,26	9,15	8,29	9,80	8,51	9,54	10,26	8,90

Table 4.19 Comparison of speedup values for different input [G] sizes on various number of processors for PSORT, PSRS, QSORT algorithms

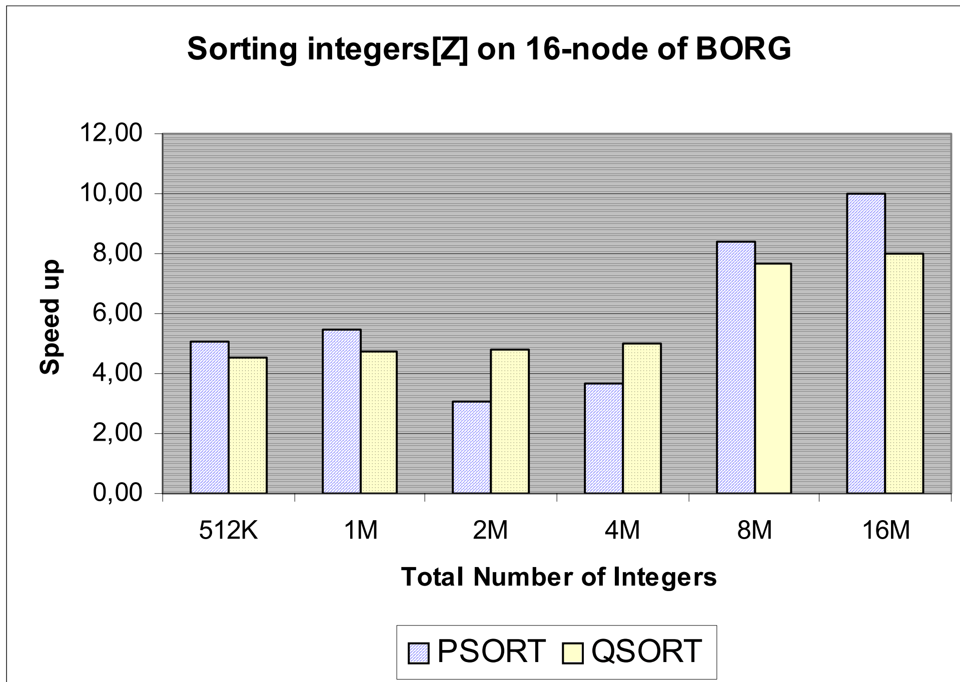


Figure 4.15 Speed up versus number of zero distributed [Z] inputs

Comparison Chart [Z]																		
Speedup = Tsequential / Tparallel																		
	input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
2	1,28	NA	1,12	1,30	NA	1,14	1,30	NA	1,16	1,33	NA	1,18	2,03	NA	1,79	1,51	NA	1,65
4	1,98	NA	1,71	1,99	NA	1,76	2,05	NA	1,81	2,06	NA	1,84	3,23	NA	2,80	3,40	NA	2,83
8	3,15	NA	2,71	3,23	NA	2,83	2,84	NA	2,75	3,44	NA	2,99	5,56	NA	4,59	5,86	NA	4,70
16	5,04	NA	4,55	5,49	NA	4,72	3,04	NA	4,81	3,64	NA	5,02	8,37	NA	7,64	10,03	NA	7,98

Table 4.20 Comparison of speedup values for different input [Z] sizes on various number of processors for PSORT, PSRS, QSORT algorithms

As seen from the results in Table 4.15, Table 4.16, and Table 4.17 for small number of processors like in 2 and 4 Hyperquicksort gives the best execution times, while PSRS follows it and lastly PSORT comes. However when the number of processors increases PSRS gives the best execution time except the Zero[Z] benchmark, as explained early regular sampling heuristics fail for that case. For uniformly and Gaussian distributed input data PSRS performs the best results, then Hyperquicksort and finally PSORT. However for the zero benchmark multiway-merge algorithm outperforms the hyperquicksort and clearly PSRS as seen in the Table 4.17.

Since, PSRS (Parallel Sorting by Regular Sampling) [19], and [7] have asymptotically optimal bounds $O(\frac{N \log N}{P})$, for sorting N numbers with P processors, the algorithm outperforms the others in practice too (see Figure 4.10, Figure 4.11). Although hyperquicksort has also theoretically good asymptotical bounds, in practice PSRS has better than it. Because the recursion in the nature of quicksort is somewhat costly and also in order to obtain a better laod balancing hyperquicksort has multiple rounds of communication. This is the generic disadvantage of the multistep parallel sorting algorithms like [2, 15, 16, 18, 40, 42, 43, 44]. Here there is a tradeoff between having a better load balance and multiple rounds of communication overhead.

The performance of multiway-merge algorithm approximates to the performances of PSRS and hyperquicksort. Although it is not as good as the other two but it is not dramatically bad either. Compared to the other results, one of the main reasons that our implementation performs less than the others is the fact that it requires two total exchange algorithm and two successive merge operations after total exchanges while PSRS has only ones. These results obeys the theoretical analysis of the algorithm, which we had found as $O(N/P \log NP)$. The multiway-merge algorithm, also performs well when the number of duplicate elements in the input sequence increases as in the zero benchmark (e.g. Figure 4.12).

When we compare the speed up values of the three algorithms for uniformly distributed data, for two processors hyperquicksort has the largest speedup value. As the number of processors increases PSRS reaches the best speed up values among three algorithms (see Table 4.18). PSRS obtained its maximum speed up for uniformly distributed data at 16 processor for input size of 2M integers as 11,32. For Gaussian distributed data [G], except for the 2M and 4M data, multiway-merge algorithm and hyperquicksort have almost nearly speed up values. What is more, for some input sizes multiway-merge algorithm has slightly better speed up values then hyperquicksort for 16 processors as seen in Figure 4.14. This small difference between speed up values of multiway-merge algorithm and hyperquicksort becomes noticeable large for zero benchmark [Z] as observed in Figure 4.15 and Table 4.20. Similarly as in the speedups, the efficiencies of the algorithms have the same ranking given with following tables Table 4.21, Table 4.22, and Table 4.23. For comparing our results with PSRS and hyperquicksort, we took the values of speedups and efficiencies in addition to the execution times for clarity.

Comparison Chart [U]																		
efficiency = speedup / # of processors																		
	input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
2	0,80	0,87	1,02	0,81	0,86	1,04	0,82	0,89	1,05	0,82	0,88	1,09	0,76	0,82	1,01	0,82	0,88	0,89
4	0,69	0,82	0,79	0,68	0,82	0,80	0,72	0,85	0,83	0,73	0,83	0,83	0,68	0,78	0,78	0,70	0,78	0,83
8	0,61	0,76	0,64	0,60	0,75	0,65	0,62	0,77	0,67	0,64	0,74	0,69	0,63	0,71	0,66	0,64	0,75	0,69
16	0,54	0,58	0,49	0,54	0,70	0,54	0,46	0,71	0,58	0,42	0,68	0,58	0,52	0,61	0,57	0,59	0,66	0,58

Table 4.21 Comparison of efficiencies for various numbers of processors on different input [U] sizes for PSORT, PSRS, QSORT algorithms

Comparison Chart [G]																		
efficiency = speedup / # of processors																		
	input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
2	0,80	0,86	1,01	0,81	0,88	1,04	0,82	0,86	1,09	0,82	0,87	1,07	0,77	0,81	1,03	0,83	0,88	0,94
4	0,68	0,78	0,80	0,70	0,79	0,82	0,70	0,75	0,82	0,72	0,80	0,83	0,68	0,75	0,79	0,70	0,75	0,79
8	0,60	0,67	0,63	0,61	0,70	0,65	0,56	0,62	0,65	0,62	0,71	0,68	0,62	0,67	0,62	0,64	0,68	0,65
16	0,57	0,60	0,52	0,56	0,63	0,54	0,35	0,60	0,56	0,42	0,64	0,57	0,52	0,61	0,53	0,60	0,64	0,56

Table 4.22 Comparison of efficiencies for various numbers of processors on different input [G] sizes for PSORT, PSRS, QSORT algorithms

Comparison Chart [Z]																		
efficiency = speedup / # of processors																		
	input size																	
	512K			1M			2M			4M			8M			16M		
#pro	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT	PSORT	PSRS	QSORT
2	0,64	NA	0,56	0,65	NA	0,57	0,65	NA	0,58	0,66	NA	0,59	1,02	NA	0,90	0,76	NA	0,83
4	0,49	NA	0,43	0,50	NA	0,44	0,51	NA	0,45	0,52	NA	0,46	0,81	NA	0,70	0,85	NA	0,71
8	0,39	NA	0,34	0,40	NA	0,35	0,35	NA	0,34	0,43	NA	0,37	0,70	NA	0,57	0,73	NA	0,59
16	0,31	NA	0,28	0,34	NA	0,29	0,19	NA	0,30	0,23	NA	0,31	0,52	NA	0,48	0,63	NA	0,50

Table 4.23 Comparison of efficiencies for various numbers of processors on different input [Z] sizes for PSORT, PSRS, QSORT algorithms

As a summary, for small number of processors like 2 and 4, hyperquicksort gives the best execution results. If the number of processors increases, PSRS has a dramatic superiority over two other algorithms in our experiments. For the zero benchmark [Z], the multiway-merge algorithm has better results than the others. Since we took the same sequential running time value for each algorithm in the experiments and all the algorithms are executed on the same machines with the same conditions, the speed up and efficiency values for the experiments are appropriate for comparisons. If sorting algorithm is needed in further works on the BORG system, we suggest to use PSRS algorithm on the average, for small number of processors like 2-4 to use hyperquicksort, and when the repeated values in the input sequence is too much (like in [Z]) then using the multiway-merge algorithm will yield better performance.

Chapter 5

5 Conclusion

Computing science is always searching to find the ways of performing task efficiently and optimally. The total elapsed time for solving a problem, number of machines used to solve it and scalability of the solutions are only a few of important criteria which considered during this big research in computing science.

The necessity of this research starts from the dawn of computer and continues still today. We believe that it could continue forever in the future, since there is no limit in humans imagination, in their creative thinking abilities, and in wishes of men. If the human being continues their life style in the way today, the wishes will never end in the future.

To achieve maximum performance, various methods, architectures, programming approaches etc. are developed during this process. *Parallel Processing* is another approach to attack problems. It was born as a result of these researches in computing and has also got its share from them. Today, we have a diverse range of parallel architectures. Many special algorithms have been developed for specific architectures to become the *best*. But the problem here is that the efficient algorithms developed for one specific machine or architecture may not be adequate for others.

In this thesis, we present another parallel sorting algorithm, which is originally [1] designed for product networks and we have adapted it to MIMD architectures. It is suitable to wide range of MIMD architectures with its share of weaknesses and strengths. The algorithm requires only two AAPC (all-to-all personalized communication) and two one-to-one communications independent from the input size. The workload is distributed amongst the processors with evenly distributed load balancing. In addition, the algorithm requires only size of 2N/P local memory for each processor in the worst case, where N is the number of items to be sorted and P is the number of processors. It runs O ($N/P \log(NP) + \tau + \sigma N/P$) time, for $N >> P$.

We have tested the algorithm on PC cluster of Bilkent University (BORG), which is a kind of distributed memory MIMD architecture. Although it may not be the *best* algorithm for parallel sorting, the multiway-merge algorithm is a feasible one with its generality to a variety of MIMD architectures, evenly distributed load balancing properties and limited regular communication steps.

In the experimental studies that are done under this work, we have selected two more algorithms which are instances of two different types of classifications in the parallel sorting [38], one from single-step algorithms and other one from multi-step algorithms. We have also implemented the PSRS algorithm that is one of the famous one among the parallel sample sort algorithms and Hyperquicksort –an example for parallel quicksort algorithm- in the BORG system. In the experimental part of the study, for each algorithm we have experiments for 3 different benchmarks [U], [G], [Z], for six different sizes of inputs [512K, 1M, 2M, 4M, 8M, 16 Millions of integers] and for five different number of processors. Totally about 270 experimental results are gathered and reported. The repetitions of the experiments are not included in this number. Among these three algorithms, PSRS reached the best performance in the experiments for Uniformly and Gaussian distributed data. Initially, for small number of processors Hyperquicksort algorithm gave better results than PSRS and multiway-merge algorithm. However on the average and for large number of processors, PSRS superiority was observed. For the Zero [Z] benchmark, the

multiway-merge algorithm is the best one amongst the other two algorithms. As a conclusion, the performances of the algorithms change depending on the number of processors, different sizes of inputs and distribution of the input streams. Thus, if sorting algorithm is needed in further works on the BORG system, we suggest to use PSRS algorithm on the average, for small number of processors like 2-4 to use hyperquicksort, and when the repeated values in the input sequence is too much (like in [Z]) then using the multiway-merge algorithm will yield better performance.

To sum up, the multiway-merge algorithm performs well on cluster technology, which becomes very popular and takes the focus of interest in Parallel Computing Science. Although it did not obtain good results as the PSRS and Hyperquicksort, it is a successful adaptation of the multiway-merge algorithm, which is valid for a various number of MIMD like architectures. What is more, as mentioned above according to the input distribution and number of processors, we obtained some cases for which the multiway-merge algorithm performs better. The empirical results that obtained from this work will constitute a sample for further studies.

Bibliography

[1] EFE K., Fernandez A. Generalized Algorithm for Parallel Sorting on Product Networks. In *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 12, December 1997, pp. 1211-1225.

[2] K. Batcher. Sorting Networks and Their Applications. In *Proc. APIPS Spring Joint Computing Conf.*, vol. 32, pp. 307-314, 1968.

[3] K. E. Batcher. On Bitonic Sorting Networks. In *Proc. 1990 International Conference on Parallel Processing*, vol. I, pp.376-379, 1990

[4] D. L. Lee, K. E. Batcher. On Sorting Multiple Bitonic Sequences. In *Proceedings 1994 International Conference on Parallel Processing,* vol. I, pp. 12-125, August 1994

[5] K. J. Lizska and K.E. Batcher. A Generalized Bitonic Sorting Network. In *Proceedings 1993 International Conference on Parallel Processing,* vol. I, pp. 105-108, 1993

[6] D. Knuth. Searching and Sorting. In *The Art of Computer Programming*, vol. 3. Reading, Mass.: Addison-Wesley, 1973.

[7] D. R. Helman, D. A. Bader, and J. JáJá. Parallel Algorithms for Personalized Communication and Sorting with an Experimental Study. Presented at the *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, pp. 211-220, June 1996.

[8] M. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, vol. 54, 1966, pp. 1901-1909

[9] H. T. Kung, C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings '78*, Academic Press, Orlando FL, 1979, pp. 256-282

[10] C. D. Thompson and H. T. Kung. Sorting on a Mesh-Connected Parallel Computer. In *Comm. ACM,* vol. 20, pp. 263-271, April 1977

[11] S. Kung, S. Lo, S. Jean, J. Hwang. Wavefront array processors – concept to implementation. In *IEEE Computer*, vol. 20, no. 7, July 1987, pp. 18-33

[12] J. Fisher. The VLIW machine: A multiprocessor for compiling scientific code. In *IEEE Computer*, vol. 17, no. 7, July 1984, pp. 34-47

[13] K. Hwang, D. Degroot. *Parallel Processing for Supercomputers & Artificial Intelligence.* McGraw-Hill, New York, 1989

[14] G.S. Akl. *Parallel Sorting Algorithms.* Academic Press,inc. 1985

[15] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann Publishers, 1992

[16] F. T. Leighton. Tight Bounds on the Complexity of Parallel Sorting, In *IEEE Transactions on Computers*, C-34, pp.344-354, 1985

[17] S. Rajasekaran. A Framework For Simple Sorting Algortihms On Parallel Disk Systems. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, June 1998

[18] V. Kumar, A. Grama, A. Gupta, Karypis George. *Introduction to Parallel Computing Design and Analysis of Algortihms.* The Benjamin/Cummings Publishing Company, 1994

[19] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. In *Parallel Computing*, Vol.19, pp.1079-1103, 1993.

[20] H. Shi and J.Schaeffer. Parallel Sorting by Regular Sampling, In *Journal of Parallel and Distributed Computing*, vol. 14, pp. 361-372, 1992

[21] N. Park, K. V. Prasanna, S. C. Raghavendra. Efficient Algorithms for Block-Cyclic Array redistribution Between Processor Sets. In *IEEE transactions on Parallel and Distributed Systems*, Vol. 10, No. 12, December 1999

[22] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Technical Report, University of Tennessee, Knoxville, TN, June 1995. Version 1.1.

[23] LAM / MPI Parallel Computing, http://www.lam-mpi.org/

[24] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI|The Complete Reference*, Volume 2, The MPI Extensions. The MIT Press, 1998.

[25] The Beowulf Project, www.beowulf.org.

[26] PVM, Parallel Virtual Machine, http://www.epm.ornl.gov/pvm/pvm_home.html

[27] S. Olariu, M. C. Pinotti, S.Q. Zheng. How to Sort N Items Using a Sorting Network of Fixed I/O Size. In *IEEE transactions on Parallel and Distributed Systems*, Vol. 10, No. 5, May 1999

[28] A. Alexandrov, M. Ionescu, K. Schauser, C. Scheiman. LogGP: Incorporating Long Messages into theLogP Model - One step closer towards a realistic model for parallel computation. In *7th annual ACM Symposium on Computer Architecture*, pp. 95-105, Santa Barbara, CA, July 1995

[29] D.E. Culler, Rm.M. Karp, D.A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993

[30] H. Stone, Parallel Processing with the Perfect Shuffle. In *IEEE Transactions Computers*, vol. 20, no.2, pp. 153-161, February 1971

[31] D. Nassimi and S. Sahni. Bitonic Sort on a Mesh-Connected Parallel Computer. In *IEEE Transactions Computers*, vol. 27, no. 1, pp.2-7, 1979

[32] F. Preparata and J. Vuillemin. The Cube-Connected Cycles: A Versalite Network for Parallel Computation. In *Comm. ACM,* vol. 24, pp. 300-309, May 1981

[33] D. Nath, S.N. Maheswari, and P. C. P. Bhatt. Efficient VLSI Networks for Parallel Processing based on Orthogonal Trees. In *IEEE Trans. Computers,* vol. 32, no. 6, pp. 569-581, June 1983.

[34] T. Nakatani, S. T. Huang, B. W. Arden, and S. K. Tripathi, K-Way Bitonic Sort. In *IEEE Transactions on Computers,* vol. 38, no. 2, pp. 283-288, February 1989

[35] Data Reorganization, www.data-re.org

[36] T. Braunl. *Parallel Programming – An Introduction*. Prentice Hall Int. Limited, 1993

[37] G. Chaudhry, T. H. Cormen, L. F. Wisniewski. Implementation of Out-Of-Core Column sort on a multiprocessor with a parallel disk system. Technical Report, Sun Microsystems, 2000

[38] H. Li. and K. C. Sevcik. Parallel Sorting by Overpartitioning. Technical Report CSRI-295, Computer Systems Research Institute, University of Toronto, Canada, April 1994.

[39] C. G. Plaxton, G. E. Belloch, C. E. Leiserson, B. M. Maggs, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 3-16, July 1991.

[40] C. G. Plaxton, Load balancing, Selection and Sorting, on the Hypercube, In *Proceedings 1989 ACM Symposium on Parallel Algorithms and Architectures,* pp. 64-73, 1989.

[41] J. S. Huang and Y. C. Chow. Parallel Sorting and Data Partitioning by Sampling. In *Proceedings of the 7th Computer Software and Applications Conference*, pp. 627-631, November 1983.

[42] B. Wagar. Hyperquicksort: A Fast Sorting Algorithm for Hyperqubes. In *Proceedings of the Second Conference on Hypercube Multiprocessors,* pp. 292-299, 1986.

[43] W. L. Hightower, J. F. Prins, and J. H. Reif. Implementations of Randomized Sorting on Large Parallel Machines. In *Proceedings of the 4th Symposium on Parallel Architectures and Algorithms,* pp. 158-167, San Diego, CA, July 1992.

[44] A. Tridgell and R. P. Brent. An Implementation of a General-Purpose Parallel Sorting Algorithm. Technical Report TR-CS-93-01, Computer Sciences Laboratory, Australian National University, Canberra, Australia, February 1993.

[45] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. MIT Press, Cambridge, Massachusetts, 1990

[46] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Efficient Algorithms for Parallel Sorting on Mesh Multicomputers. *International Journal of Parallel Programming,* pp. 95-131.

[47] C. U. Martel and D. Q. Gusfield. A Fast Parallel Quicksort Algorithm. *Information Processing Letters,* pp. 97-102, 1989.

[48] G. C. Fox, M. Johnson, G. Lyzenga, S. W. Otto, J. Salmon and D. Walker. *Solving Problems on Concurrent Processors: Vol.1.* Prentice Hall, Englewood Cliffs, NJ, 1988.

6 Appendices

Input size	512K = 2^{19}	1M = 2^{20}	2M = 2^{21}	4M = 2^{22}	8M = 2^{23}	16M = 2^{24}
# of proc	524288	1048576	2097152	4194304	8388608	16777216
1	1,331556	2,82145	5,937117	12,457858	23,860795	51,60693
1	1,340341	2,81894	5,938312	12,453154	24,010314	49,953322
1	1,33595	2,820809	5,936176	12,452075	23,878915	49,833601
1	1,33332	2,825863	6,014676	12,460992	23,877341	50,217576
1	1,33389	2,815444	6,001336	12,452153	24,547551	51,433751
Avg	1,3350114	2,8205012	5,9655234	12,4552464	24,0349832	50,40285725
2	0,84815	1,740235	3,620729	7,63464	15,748585	30,824214
2	0,83404	1,740065	3,621162	7,527606	15,836642	30,730728
2	0,846087	1,739822	3,624371	7,528154	15,863817	30,717127
2	0,835047	1,73788	3,6202	7,538016	15,880906	30,716324
2	0,834804	1,741542	3,62464	7,543892	15,676042	30,505195
Avg	0,8396256	1,7399088	3,6222204	7,5544616	15,8011984	30,75735633
4	0,484829	1,03269	2,047932	4,245199	8,795382	18,504009
4	0,489565	1,060532	2,04696	4,249882	8,776898	18,201944
4	0,482546	1,009541	2,102681	4,238162	8,959687	17,867881
4	0,48452	1,026937	2,116242	4,281864	8,880975	17,797193
4	0,48062	1,054527	2,114022	4,279668	8,883335	17,738417
Avg	0,484416	1,0368454	2,0855674	4,258955	8,8592554	18,0218888
8	0,275967	0,587836	1,201419	2,346242	4,879598	9,74025
8	0,269595	0,583133	1,206666	2,482858	4,824711	9,844373
8	0,277492	0,590626	1,186264	2,368265	4,783823	9,82371
8	0,266146	0,601544	1,258594	2,524292	4,761079	9,817347
8	0,278174	0,586932	1,171126	2,373236	4,776463	9,724531
Avg	0,2734748	0,5900142	1,2048138	2,4189786	4,8051348	9,7900422
16	0,160118	0,320247	0,725158	1,663814	2,86068	5,421947
16	0,158398	0,336058	0,7431	1,610893	2,903732	5,349319
16	0,15337	0,321306	0,668518	1,903183	2,921287	5,40896
16	0,150118	0,312463	0,950699	1,74597	2,995499	5,361329
16	0,148735	0,340381	0,981341	2,238277	2,878135	5,297552
Avg	0,1541478	0,326091	0,8137632	1,8324274	2,9118666	5,3678214

Table 6.1 Data obtained from the experiments for sorting *integers* [U] on *BORG* with PSORT.

Input size	512K = 2^{19}	1M = 2^{20}	2M = 2^{21}	4M = 2^{22}	8M = 2^{23}	16M = 2^{24}
# of proc	524288	1048576	2097152	4194304	8388608	16777216
1	1.311402	2.826437	5.847682	12.301117	23.784586	50.331247
1	1.311304	2.775577	5.855555	12.289182	24.09196	50.733111
1	1.312682	2.771098	5.850357	12.301611	23.919278	50.733111
1	1.310748	2.780947	5.853703	12.288133	23.941938	49.779805
1	1.310174	2.866625	5.850753	12.297429	23.947187	49.543677
Avg	1.311262	2.8041368	5.85161	12.2954944	23.9369898	50.3943185
2	0.823266	1.719904	3.591976	7.565149	15.551137	30.544635
2	0.823755	1.720953	3.589323	7.566341	15.728051	30.533321
2	0.825109	1.720236	3.591642	7.562311	15.646591	30.503839
2	0.824635	1.720645	3.586868	7.514876	15.654865	30.530064
2	0.823462	1.720477	3.585335	7.475842	15.547923	30.684456
Avg	0.8240454	1.720443	3.5890288	7.5369038	15.6257134	30.527265
4	0.489234	0.977203	2.052974	4.253341	8.852495	17.760773
4	0.477921	0.998674	2.103432	4.229291	8.665496	17.962157
4	0.474659	1.006689	2.12547	4.363844	8.878012	17.920993
4	0.486426	1.054054	2.053381	4.199804	8.871281	17.947053
4	0.483754	1.004461	2.056086	4.276828	8.648423	18.163126
Avg	0.4823988	1.0082162	2.0782686	4.2646216	8.7831414	17.9508204
8	0.266984	0.562115	1.176946	2.525738	4.899292	9.827816
8	0.28269	0.575803	1.354918	2.504136	4.880004	9.773595
8	0.276173	0.571787	1.281885	2.473964	4.883803	9.754557
8	0.272937	0.576859	1.429068	2.405257	4.746033	9.926793
8	0.27583	0.593461	1.321335	2.543729	4.803721	9.718709
Avg	0.2749228	0.576005	1.3128304	2.4905648	4.8425706	9.800294
16	0.1387	0.307564	1.003127	1.650454	2.998168	5.245192
16	0.156937	0.304306	1.072952	1.999165	2.959577	5.264493
16	0.147562	0.311357	1.111057	2.168219	2.765706	5.284548
16	0.140498	0.326699	1.049302	1.579132	2.852557	5.30288
16	0.137071	0.313825	1.054444	1.68718	2.852826	5.309893
Avg	0.1441536	0.3127502	1.0581764	1.81683	2.8857668	5.2814012

Table 6.2 Row data obtained from the experiments for sorting *integers* [G] on *BORG with PSORT*.

Input size / # of proc	512K = 2^{19}	1M = 2^{20}	2M = 2^{21}	4M = 2^{22}	8M = 2^{23}	16M = 2^{24}
	524288	1048576	2097152	4194304	8388608	16777216
1	0,607428	1,281447	2,671633	5,544792	17,370429	35,99424
1	0,609549	1,273434	2,676843	5,563932	17,380735	35,988456
1	0,619182	1,286043	2,691106	5,562792	17,390156	35,989307
1	0,606327	1,278813	2,669899	5,572786	17,380198	35,991249
1	0,606102	1,270363	2,667436	5,560017	17,144863	35,901343
Avg	0,6097176	1,27802	2,6753834	5,5608638	17,3332762	35,990813
2	0,477157	0,984349	2,022297	4,225383	8,512896	23,822524
2	0,477775	0,979606	2,112916	4,244817	8,514749	23,81894
2	0,475768	0,999785	2,053412	4,150021	8,629512	23,820873
2	0,478391	0,984213	2,089066	4,144266	8,498519	24,018429
2	0,476467	0,980351	2,023062	4,159833	8,505883	25,198367
Avg	0,4771116	0,9856608	2,0601506	4,184864	8,5323118	23,820779
4	0,311054	0,637339	1,280608	2,686521	5,313281	10,411763
4	0,30511	0,656613	1,347829	2,627289	5,261572	10,589217
4	0,309205	0,643719	1,29162	2,727654	5,43301	10,69407
4	0,312957	0,644722	1,324167	2,661194	5,507351	10,580495
4	0,303339	0,631461	1,282935	2,776059	5,306571	10,710446
Avg	0,308333	0,6427708	1,3054318	2,6957434	5,364357	10,5971982
8	0,187302	0,393551	1,089147	1,624537	3,14829	6,361053
8	0,192947	0,389531	1,027092	1,584266	3,13872	6,243545
8	0,197088	0,401502	0,972976	1,637642	3,123051	6,025125
8	0,195316	0,382138	0,810717	1,588441	3,077117	6,133831
8	0,195063	0,410747	0,81548	1,640392	3,088518	5,94932
Avg	0,1935432	0,3954938	0,9430824	1,6150556	3,1151392	6,1425748
16	0,117326	0,225012	0,869158	1,304136	2,147669	3,541467
16	0,125695	0,250186	0,890927	1,72667	1,94073	3,54921
16	0,128421	0,243825	0,882188	1,497912	2,115439	3,581412
16	0,106873	0,224691	0,899128	1,37998	2,071702	3,646695
16	0,126956	0,220288	0,861114	1,73018	2,075408	3,623104
Avg	0,1210542	0,2328004	0,880503	1,5277756	2,0701896	3,5883776

Table 6.3 Row data obtained by the experiments for sorting *integers* [Z] on *BORG with PSORT*.