FAST PARALLEL ALGORITHMS FOR SORTING AND MEDIAN FINDING

Taoufik Dachraoui

A thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

November 1996

0-612-26012-7

Abstract

Fast Parallel Algorithms for Sorting and Median Finding

Taoufik Dachraoui

The problem of sorting has been studied extensively in the literature because of its many important applications and because of its intrinsic theoretical significance. Efficient sorting algorithms for parallel machines with some fixed processor interconnection pattern are relevant to almost any use of parallel machines. There have been many efforts to find the best algorithm for sorting on different parallel machines; Blelloch *et al.* [15] provided a quantitative analysis of the circumstances under which different algorithms for sorting on the CM-2 were advantageous, and a similar effort was made by Hightower *et al.* [40] on Maspar's MP-1, by Dickmann *et al.* for a Parsytec GCel [24], and by Helman *et al.* on a number of platforms [38].

Many sorting algorithms that perform well on uniformly distributed data suffer significant performance degradation on non-random data. Unfortunately many real-world applications require sorting on data that is not uniformly distributed. In this thesis, we propose a new strategy, *A-ranksort*, for sorting on parallel machines which has a stable behavior on input distributions of different entropies. Our sorting algorithm essentially consists of deterministically computing approximate ranks for the input keys, and using these to route to destinations that are guaranteed to be close to the final destination, and finally using a few steps of odd-even transposition sort to complete the sorting.

We implemented *A-ranksort*, *B-flashsort* [40], *Radix sort* [20], and *Bitonic sort* [6] on a 2048 processor Maspar MP-1. Thearling [73] proposes a test suite of inputs with which to evaluate the performance of sorting algorithms. We tested all the algorithms on a very similar test suite. Our experiments show that *A-ranksort* outperforms all the other algorithms on a variety of input distributions, when the output is required to be evenly distributed over the processors.

Using similar ideas, we also designed a new deterministic selection algorithm for integers that is simple and fast. Our experiments show that our selection algorithm is up to 25 times faster than *Radix sort* and up to 5.3 times faster than *B-flashsort* implemented on MasPar MP-1. In contrast, the best known comparison-based selection algorithm [14] on the TMC CM-5 is less than 3 times faster than *Radix sort.*

Acknowledgments

I would like to thank Dr. Lata Narayanan who has contributed significantly to the quality of this thesis. The work on this thesis could not have been started if it were not for her generous support. Together, we wrote a paper on the parallel sorting algorithm *A-ranksort* that appeared in the *Proceedings of the IEEE Symposium on Parallel and Distributed Processing* 1996 (SPDP'96) [25].

Also, I would like to thank the Parallel Processing Group at the *Centre de Recherche Informatique a Montreal* (CRIM) for their support and help to use their MasPar MP-2 parallel machine, and the Department of Computer Science at the Laval University for their support and help to use their MasPar MP-1 parallel machine on which we could implement and test all the algorithms presented in this thesis. I also thank Dr. J. Prins at the University of North Carolina at Chapel Hill for allowing us to use his implementation of *B-flashsort* that we found faster than our implementation of the same algorithm.

I thank all the members of my family for their love and support during the work on this thesis. The love, patience, and encouragement of my family made this thesis possible. I affectionately dedicate this thesis to them.

Finally, I thank God the unique, the most merciful, and the most powerful, without His guidance and His help nothing is possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The design and analysis of parallel algorithms involve a complex set of inter-related issues that is difficult to model appropriately. These issues include computational concurrency, processor allocation and scheduling, communication, synchronization, and granularity (granularity is a measure of the amount of computation that can be performed by the processors between synchronization points). An attempt to capture most of the related parameters makes the process of designing parallel algorithms a challenging task.

It is widely recognized that an important ingredient for the success of parallel processing is the emergence of computational models that can be used for design and analysis of algorithms and for accurately predicting the performance of these algorithms on real machines [68].

1.1 Parallel Processing

Parallel and distributed computation is currently an area of intense research activity, motivated by a variety of factors. There has always been a need for the solution of very large computational problems. Technological advances have raised the possibility of massively parallel computation and have made the solution of such problems possible.

Parallel computing systems [1, 77] consist of several processors that are located within a small distance of each other. Their main purpose is to execute jointly a computational task and they have been designed with such a purpose in mind; communication between processors is reliable and predictable. Distributed computing systems [58, 77] are different in a number of ways. Processors may be far apart, and interprocessor communication is more problematic. Communication delays may be unpredictable, and the communication links themselves may be unreliable. Furthermore, the topology of a distributed system may undergo changes while the system is operating, due to failures or repairs of communication links, as well as due to addition or removal of processors. Distributed computing systems are usually loosely coupled; there is very little, if any, central coordination and control.

We are interested in computers that use high degree of parallelism to speed the computation required to solve a single large problem. This leaves out distributed systems such as a network of personal workstations, because, although the number of processing units can be quite large, the communication in such systems is currently too slow to allow close cooperation on one job.

The main purpose of parallel processing is to perform computations faster than can be done with a single processor by using a number of processors concurrently. The pursuit of this goal has had a tremendous influence on almost all the activities related to computing. The need for faster solutions and for solving larger-size problems arises in a wide variety of applications. These include fluid dynamics, weather prediction, modeling and simulation of large systems, information processing and extraction, image processing, artificial intelligence, and automated manufacturing.

Three main factors have contributed to the current strong trend in favor of parallel processing. First, the hardware cost has been falling steadily; hence, it is now possible to build systems with many processors at a reasonable cost. Second, the very large scale integration (VLSI) circuit technology has advanced to the point where it is possible to design complex systems requiring millions of transistors on a single chip. Third, the fastest cycle time of a von Newmann-type processor seems to be approaching fundamental physical limitations beyond which no improvement is possible; in addition, as higher performance is squeezed out of a sequential processor, the associated cost increases dramatically. All these factors have pushed researchers into exploring parallelism and its potential use in important applications [77].

Parallel computers can be classified according to a variety of architectural features and modes of operations [28, 32, 34, 37]. In particular, these criteria include the type and the number of processors, the interconnections among the processors and the corresponding communication schemes, the overall control and synchronization, and the input/output operations. For more information on parallel computing see [1, 42, 55, 77].

1.2 Models of Computation

Computational models that can be used for design and analysis of algorithms and for accurately predicting the performance of these algorithms on real machines should be a "bridging model" that links the two layers of hardware and software [76, 77].

A universally accepted model for designing and analysing sequential algorithms consists of a central unit with a random access memory attached to it. The success of this model is primarily due to its simplicity and its ability to capture in a significant way the performance of sequential algorithms on von Newmann-type computers [22, 60]. Unfortunately, parallel computation suffers from the lack of a commonly accepted model due to the additional complexity introduced by the presence of a set of interconnected processors. In particular, the performance of a parallel algorithm seems to depend on several factors such as overall resource requirements, computational concurrency, processor allocation and scheduling, communication, and synchronization.

There is a variety of models of parallel computation, incorporating different assumptions on the computing power of each individual processor and on the interprocessor information transfer mechanism.

Algorithmic models typically supply a simple abstraction of a computing device and a set of primitive operations assumed to execute in a fixed *unit time*. The assumption that primitives operate in unit time allows researchers to greatly simplify the analysis of algorithms, but is never strictly valid on real machines: primitives often execute in time dependent on machine and algorithm parameters. For example, in the serial random access machine (RAM) model [26], memory references are assumed to take unit time even though the data must fan-in on real hardware and therefore take time that increases with memory size. In spite of this inaccuracy in the model, the unit-time assumption has served as an excellent basis for the analysis of algorithms.

In the parallel random access machine (PRAM) models [30, 31, 33, 53, 66, 71], memory references are again assumed to take unit time. For parallel algorithms that access data in an arbitrary fashion, the assumption of unit-time memory references can be justified. Accessing data through the network is a relatively slow operation in comparison with arithmetic and other operations. Thus, counting the number of parallel memory accesses executed by two parallel algorithms does, in fact, yield a fairly accurate estimate of their relative performance.

However, the running time estimate is not as accurate as in RAM model. The principal way in which real machines violate the unit-time abstraction of the PRAM model is that some memory-access patterns are faster than others; Typically, real parallel machines have a communication network that can support the abstraction of a global memory. The algorithms designed in the PRAM model have a high degree of parallelism [42] but perform a great deal of inter-processor communication. This leaves unresolved the question of how to implement algorithms designed for a PRAM model on real machines which have limited inter-processor communication bandwidth.

Many other models of parallel computation (e.g. BSP [76], LogP [19], and BDM [43] models) were developed by researchers to address the limitations of the PRAM model and gain improved performance measures. All these models are essentially extensions to the basic PRAM model.

In practice, the user usually has a specific parallel machine of fixed size and wants to find the best algorithm to use on that particular machine. Usually the size of the data set is much larger than the number of processors in the machine, and it is important to consider the scalability of parallel algorithms. A different approach is to use a parameterized model, that has as parameters costs for various operations. Implementations of the algorithm can then confirm the predictive capability of the analytic model.

1.3 Sorting and Selection

Sorting is one of the most common operations performed by a computer and one of the best-studied problems in computer science. Because sorted data are easier to manipulate than randomly-ordered data, many algorithms require sorted data. An early but exhaustive compilation of sorting algorithms and their behavior can be found in [45]. The reason for the continuing interest in sorting is evident when we consider the wide variety of applications to this problem, ranging from database management to computational geometry. Sorting has been studied widely in the parallel computation setting as well. Apart from the applications mentioned above, sorting algorithms can be useful in solving routing problems in parallel machines. Sorting has also frequently been used as a benchmark for evaluating the performance of supercomputer architectures on non-numeric applications [68].

The problem of sorting a sequence X is the process of rearranging the elements of X such that they appear in nondecreasing or nonincreasing order. In sequential sorting algorithms, the input and the sorted sequences are stored in the processor's memory. However, in parallel sorting a particularly useful approach is that the input and the sorted elements are distributed among the processors. A general method of distribution is to order the processors and use this ordering to specify a global ordering for the sorted sequence. For instance, if P_i (processor i) comes before P_j in the enumeration, no element stored in P_i will be greater than any element stored in P_j in the output sequence. We can enumerate the processors in many ways. For certain parallel algorithms and interconnection networks, some enumerations lead to more efficient parallel formulations than others (Kunde [47, 48] showed that sorting on mesh-connected networks is slower for some indexing functions).

Sorting algorithms can be categorized as *comparison-based* and *noncomparison-based*. A comparison-based algorithm sorts an unordered sequence of elements by repeatedly comparing pairs of elements and, if they are out of order, exchanging them. This fundamental operation of comparison-based sorting is called *compare-exchange*. The sequential complexity of comparison-based sorting algorithm is $\Theta(n \log n)$, where n is the number of elements to be sorted (e.g. *Mergesort*). Algorithms that are not comparison-based sort by using certain known properties of the elements (such as their binary representation or their distribution). The lower bound complexity of these algorithms is $\Omega(n)$ (e.g. *Radix sort*).

In this thesis, we used MasPar MP-1 parallel machine that is based on a mesh topology. The mesh is an attractive topology for parallel computers chiefly on account of its simplicity and inexpensiveness. Clearly, sorting on a two-dimensional p processor square mesh requires $\Omega(\sqrt{p})$ communication steps. The first $\Theta(\sqrt{p})$ sorting algorithms on the mesh were due to Thomson and Kung [74] and Schnorr and Shamir [70]. Since then, there have been several new algorithms for sorting on the mesh [51, 49, 50, 54], each reducing the multiplicative constant in the leading term. Some of these algorithms are optimal in the sense that they can be proved to require less than $2\sqrt{p} + o(\sqrt{p})$ communication steps, when each processor contains a single input element. These algorithms have the disadvantage that they are quite complicated and the lower order term turns out to be quite expensive in practice. Additionally, we are interested in the more practical situation when N, the size of the data set, is much larger than p, the number of processors available ($N \gg p$). A straightforward transformation of these algorithms to handle this situation would then multiply the cost by a factor of $\lceil N/p \rceil$.

There have been many efforts to find the best algorithm for sorting on parallel machines; Blelloch *et al.* [15] provided a quantitative analysis of the circumstances under which different algorithms for sorting on the CM-2 were advantageous, and a similar effort was made by Hightower *et al.* [40] on Maspar's MP-1, by Dickmann *et al.* for a Parsytec GCel [24], and by Helman *et al.* on a number of platforms [38].

Another common problem performed by a computer is that of finding the median element in a set of data. The more general *selection problem* can be specified as follows: Given a set X of N numbers and a number s, where $1 \leq s \leq N$, find the element with rank s in X.

One can, of course, select the sth element by sorting the input and indexing the sth element of the output. With no assumptions about the input distribution, this method runs in $\Omega(N \log N)$ time. But if we use *Radix sort*, a non-comparison sorting algorithm, and bucket sort is used as the intermediate sorting algorithm, the selection problem is solved in $O(Nk)$ time, where the input is a set of N integers in the range $0..N^k - 1$, but the cost of the sorting increases as the size of the integers to be sorted increases.

A fast linear expected time algorithm for selection is due to Hoare [36]. An improved linear expected time algorithm that partitions around an element recursively selected from a small sample of the elements is due to Floyd and Rivest [29]. Blum *et al.* [13] showed that the selection problem can be performed in linear time in the worst case. No more than $5.4305N$ are required to select the sth smallest of N numbers, for any s, $1 \leq s \leq N$ [13].

Previous parallel algorithms for selection ([12, 35, 42, 62, 69]) tend to be network dependent or assume the PRAM model. Efficiency and portability to current parallel machines are difficult to achieve. In [14], the authors describe a comparison-based parallel selection algorithm that is motivated by similar sequential ([20]) and parallel ([42]) algorithms. They used the Block Distributed Memory (BDM) model [43, 44] as a computation model for developing and analysing their algorithm on distributed memory machines. The selection algorithm that we describe in this thesis uses the binary representation of the input elements, while all the other known algorithms for selection are based on comparing pairs of elements and they must make at least $N-1$

comparisons [45].

1.4 Scope of Thesis

In this thesis, we describe a new strategy for sorting. Our method essentially consists of computing approximate ranks for elements, and using these to route to destinations that are guaranteed to be close to the final destination. The main idea of our algorithm is to divide the keys into small segments that are ranked relative to each other. The actual ranks within the segments will be calculated at the end. The calculation of these approximate ranks is done deterministically. Our algorithm is not a comparison-based algorithm; we use the bitwise representation of keys to aid in finding approximate ranks. When these ranks are computed, keys are sent to approximate destinations which are guaranteed to be close to the final destinations. No keys are moved until the approximate ranks of keys are calculated.

Our algorithm was implemented on a 2048-processor Maspar MP-1 machine. Thearling [73] proposes a test suite of inputs with which to evaluate the performance of sorting algorithms. We tested *A-ranksort* on a very similar test suite: the results are presented in Chapter 3. We also implemented *B-flashsort* [40], *Bitonic sort* [6], and *Radix sort* [20]. In [40], *B-flashsort* was shown to outperform all the other algorithms for randomly chosen data. Our algorithm improves slightly on the performance of *B-flashsort* for uniformly distributed data, for the case when the sorting is required to be *balanced*, that is, when each processor is required to contain the same number of keys at the end.

However, few real-world applications involve uniformly distributed data [75], and algorithms that work well on random data often perform poorly on non-uniform data. Our experiments show that on many input distributions, the space requirements of *B-flashsort* simply exceed system limits. Regardless of the space limits, the running time of *B-flashsort* degrades significantly on many input distributions owing to the high load imbalance. Our algorithm, however, is relatively indifferent to the input distribution, and is the best choice for a variety of input distributions. Specifically, when the entropy of the data distribution is small, or when the data is chosen from a sparse set, our algorithm significantly outperforms *B-flashsort* as well as the other algorithms.

The advantages of our algorithm apart from its being the most efficient of known balanced algorithms are that it is conceptually simple, deterministic, and exhibits good performance for a wide variety of input distributions. It can be implemented on any parallel machine; it can work very efficiently in any machine where there is a cheap primitive for performing parallel prefix operations. Our algorithm can also be adapted to work for *any* sorting order, though our implementation is for row-major order and the performance is likely to degrade for some sorting orders. Our algorithm is based on an approximate ranking approach, which can be easily adapted to perform approximate selection. Finally, we are able to give both worst-case and average-case analysis of the algorithm in terms of the costs of various operations. Therefore, it should be straightforward to predict its performance on different parallel machines. In fact, we show that the predicted time is quite close to the empirically observed time in Section 3.3.

For the problem of selection, we designed a new deterministic algorithm, *A-Select*, for integer data that is simple and fast. *A-select*, as in *A-ranksort*, relies on the representation of keys as b-bit integers, and examines the input keys r bits at a time, starting with the most significant block of r bits in each key. The algorithm proceeds in rounds or iterations; each iteration divides the input keys into buckets, and only the keys in a selected bucket will participate in the next iteration.

An important characteristic of *A-Select* is that there is no data movement. We implemented our algorithm on MasPar MP-1. Our experimental results show that the relative performance of *A-Select* with *Radix sort* implemented on MasPar MP-1 is much better than the relative performance of the selection algorithm in [14] with *Radix sort* implemented on TMC CM-5. We predict that our algorithm would have a better performance than the comparison-based selection algorithm in [14] on the

TMC CM-5.

The thesis is organized as follows. The next chapter describes the computational models that will be used for the analysis of the algorithms and for accurately predicting the performance of these algorithms on MasPar MP-1. In Chapter 3 we describe our new sorting algorithm, and provide a performance analysis with experimental results. Empirical comparisons with previously known sorting algorithms are provided in Section 3.5. In Chapter 4 we propose a new fast and deterministic selection algorithm that uses the bitwise representation of the keys. We discuss future directions in Chapter 5.

Chapter 2

Model of Computation

It is widely recognized that an important ingredient for the success of parallel processing is the emergence of computational models that can be used for design and analysis of algorithms and for accurately predicting the performance of these algorithms on real machines. A major challenge is to move toward architectures that can efficiently implement a truly general-purpose parallel computation model [68]. The computation model should be a "bridging model" that links the two layers of hardware and software [76].

2.1 Parallel Computer Architectures

Traditional sequential computers are based on the model introduced by John von Neumann [60]. This computational model takes a single sequence of instructions and operates on a single sequence of data. Computers of this type are often referred to as *single instruction stream, single data stream* (SISD) computers.

There are many ways in which parallel computers can be constructed. These computers differ along various dimensions such as control mechanism, address-space organization, interconnection network, and granularity of processors.

The most widely used classification of parallel computational models is the simple one proposed by Flynn [27], where essentially two different control mechanisms are proposed and nowadays widely used for parallel computers. These are *single instruction stream multiple data stream* (SIMD), and *multiple instruction stream multiple data stream* (MIMD) architectures.

In SIMD architectures, a single control unit dispatches instructions to each processing unit, that is, the same instruction is executed synchronously by all processing units. Processing units can be selectively switched off during an instruction cycle. SIMD computers use simple and readily scalable hardware to implement data parallelism, which is a simple programming model. Examples of SIMD parallel computers include the Illiac IV, MPP, DAP, CM-2, MasPar MP-1, and MasPar MP-2.

MIMD architecture is a more general design capable of performing well over a broader range of applications, where the processors each have their own control flow and are inherently asynchronous. However, some new concerns, that do not exist in SIMD architectures, must be addressed. Some of these serious issues are the increased complexity of programmability and debugging of MIMD algorithms caused in part by the asynchronous nature of the MIMD machine. This inherent asynchrony means that the set of all possible interleaving of the instructions is very large, and that the designers of MIMD algorithms must consider all possible total orderings of the operations and include sufficient coordination commands to ensure that only proper interleavings can occur. Also, hardware deadlocks can occur with MIMD architectures but not with SIMD architectures, thus special attention is required to avoid such problems.

In SIMD computers the processors are kept in lockstep by the broadcast instructions. Therefore, unlike the MIMD computers, the processors need not communicate to each other for synchronization purposes. Also, SIMD computers require less hardware than MIMD computers because they have only one global control unit, thus the processors can be smaller and more numerous. Furthermore, in MIMD computers, each processor needs to store a copy of the program and the operating system, while in SIMD computers only the global control unit needs to do so.

Another important aspect of parallel architecture is the organization of the memory system. Generally speaking, there are two extreme alternatives known as *shared memory* and *distributed memory* or *message-passing* architectures, and a variety of hybrid designs lying in between [37, 77]. In *shared memory* architecture there is one large global memory, and all processors have equal access to data and instructions in this memory. A processor can communicate to another by writing into the global memory, and then having the second processor read that same location in the memory. This solves the interprocessor communication problem, but introduces the problem of simultaneous accessing of different locations of the memory by several processors. In *distributed memory* architecture each processor has a *local memory* that is not accessible from any other processor.

Also, an important design issue in parallel processor and multicomputer systems is the type and topology of the *interconnection network*. A network is used to connect processors or computers together or processors to memory. The availability of more efficient and reliable networks is essential for achieving high performance. Many network topologies have been proposed in the literature, for example, ring networks, meshes, shuffle exchanges networks, and hypercubes [55, 67].

The diverse architectures and interconnection networks of the parallel machines have a pervasive impact on algorithm performance. Each architecture has distinct properties on which may depend the performance of algorithms.

2.2 Survey of Models of Computation

A universally accepted model for designing and analysing sequential algorithms consists of a central unit with a random access memory (RAM) attached to it. The success of this model is primarily due to its simplicity and its ability to capture in a significant way the performance of sequential algorithms on von Newmann-type computers [22, 60].

Unfortunately, parallel computation suffers from the lack of a commonly accepted model due to the additional complexity introduced by the presence of a set of interconnected processors. Although much of human cognition does take place in a parallel way, it seems that humans conscious thinking is sequential and that makes designing, understanding and debugging parallel programs difficult. In particular, the performance of a parallel algorithm seems to depend on several factors such as overall resource requirements, computational concurrency, processor allocation and scheduling, communication, and synchronization.

An algorithm designer should identify the architectural features that significantly affect the performance of a parallel machine, and assign to each feature a quantitative or qualitative measure (*resource metric*). These resource metrics abstract the architectural details of the parallel machine. Typical resource metrics include the number of processors, communication latency, bandwidth, block transfer capability, network topology, memory hierarchy, memory organization, and degree of asynchrony [56].

Historically, the Parallel Random Access Machine (PRAM) is the most widely used parallel model [30]. The PRAM model is the natural parallel analog of the von Newmann model. In its simplest form the PRAM model consists of p synchronous processors all operating and communicating via a shared memory. The number of processors is usually a function of the input of a problem to be solved. The main difference among PRAM models is in how they deal with the access of many processors to the same memory cell if it is allowed [42, 77]. A PRAM computation is a sequence of read, write (to the shared memory) and local computation steps, where the costs of the memory access and local computation steps are uniform, and there is no extra cost for synchronization.

Addressing the limitations of the PRAM model has motivated the development of other models of parallel computation: the BSP model [76], the LogP model [19], the BDM model [43], and recently the PRAM(m) model [59]. All these models are extensions to the basic PRAM model. The extensions may be viewed as adding more resource metrics to the PRAM model in order to gain improved performance measures.

Another solution to address the limitations of the idealistic model PRAM was suggested by Blelloch in [8]: to add other primitives to the PRAM models that can execute as fast as memory references in practice, and that can reduce the number of program steps of algorithms, therefore making the algorithms more practical. In his paper, Blelloch studied the effects of adding two scan primitives as unit-time primitives to the PRAM models. The study showed that these scan primitives take no longer to execute than parallel memory references, both in practice and in theory, but yet can improve the asymptotic running time, and the description of many algorithms.

In this thesis we use a mesh-connected network model to which we add three primitives, the *scan*, the *reduceAdd* and the *route* operations. However, in our model the primitive operations are not assumed to take unit-time, instead they are machine-dependent and their time is measured empirically.

2.3 Experimental Environment

The first massively parallel machines had SIMD designs. This initial interest in SIMD resulted both from characteristics of early parallel applications and economic necessity. Commercial development and sales of SIMD machines for somewhat more general applications is continuing. We implemented all the algorithms described in this thesis on the MasPar MP-1 machine with 2048 processor elements.

2.3.1 MasPar MP-1 Characterization

MasPar Computer Corporation has designed and implemented a high performance, low-cost, massively parallel computing system called the MP-1. The system works in a SIMD fashion [9].

The MasPar MP-1 parallel processing system is a massively data parallel processing system. It consists of a front end and a data parallel unit (DPU). The front end supports a workstation that runs the ULTRIX operating system and standard I/O. The DPU consists of an array control unit (ACU), an array of processor elements

(PEs) (up to 16,384 PEs), and PE communications mechanisms. The DPU is where all parallel processing is done. The ACU is a processor with its own registers and data and instruction memory. It controls the PE array and performs operations on singular data (see below). The ACU sends data and instructions to each PE simultaneously. Each PE is a load/store arithmetic processing element with dedicated registers and RAM. Each PE receives the same instruction simultaneously from the ACU. PEs that are active (see below) execute the instruction on variables that reside on the PEs.

The aspects of MasPar MP-1 that are important for our purposes are:

- The parallel portion of the machine consists of a fully-programmable controller and an array of PEs (up to 16,384).

- The PE's are arranged on a 2-D torus, with each processor connected directly to its eight nearest neighbors. Short distance regular communication is very efficient using this network.

- A global router network that permits random processor-to-processor communication using a circuit-switched, hierarchical crossbar communications network.

- Two global buses: a common bus on which the ACU broadcasts instructions and data to all or selected processors, and a logical OR-tree which consolidates status responses from all the processors back to the ACU.

- Each processor has full addressing autonomy within its local memory. However, an indirect memory access can take approximately 3 times longer than a direct memory access (same memory cell address at all PE's).

For a detailed description of MasPar MP-1 see [9, 17, 61]. Approximate timings for the operations listed above are given in Table 2.

2.3.2 Programming Model: MPL Language

One of the most notable advances in computing technology over the past decade has been in the use of parallelism, or concurrent processing, in high-performance computing. Of the many types of parallelism, two are most frequently cited as important to modern programming [1]:

- *control parallelism*, which allows two or more operations to be performed simultaneously. (Two well-known types of control parallelism are *pipelining*, in which different processors, or groups of processors, operate simultaneously on consecutive stages of a program, and *functional parallelism* in which different functions are handled simultaneously by different parts of the computer. One part of the system, for example, may execute an I/O instruction while another does computation, or separate addition and multiplication units may operate concurrently. Functional parallelism frequently is handled in the hardware: programmers need take no special actions to invoke it.)
- *data parallelism*, in which more or less the same operation is performed on many data elements by many processors simultaneously.

While both control and data parallelism can be used to advantage, in practice the greatest rewards have come from data parallelism. There are two reasons for this.

First, data parallelism offers the highest potential for concurrency. Each type of parallelism is limited by the number of items that allow concurrent handling: the number of steps that can be pipelined before dependencies come into play, the number of different functions to be performed, the number of data items to be handled. Since in practice the last of these three limits is almost inevitably the highest (being frequently in the thousands, millions, or more), and since data parallelism exploits parallelism in proportion to the quantity of data involved, the largest performance gains can be achieved by this technique.

Second, data parallelism code is easier to write, understand, and debug than control parallel code. The reasons for this are straightforward. Data parallel languages are nearly identical to standard serial programming languages. Each provides some method for defining parallel data structures. Very little new syntax is added: the power of parallelism arises simply from extending the meaning of existing program syntax when applied to parallel data.

The flow of control in a data parallel language is also nearly identical to that of its serial counterpart. Since this control flow, rather than processor speed, determines the order of execution, race conditions and deadlock cannot develop. The programmer does not have to add extra code to ensure synchronization within a program; the compilers and other system software maintain synchronization automatically. Moreover, the order of events, being essentially identical to that in a serial program, is always known to the programmer, which eases debugging and program analysis considerably.

All the algorithms in this thesis are implemented using the MPL language. The MPL [57] is a data-parallel language based on C as defined in the ANSI standard (ANSI x3.159 - 1988). A description of the added features which provide parallel processing follows.

- A new keyword, *plural*, distinguishes between two independent address spaces. Variables defined using the keyword *plural* are located identically on each PE in the PE array. All the other variables are singular variables and are allocated on the ACU.
- Plural expressions are supported.
- All arithmetic and addressing operations defined in ANSI C are supported for plural data types. For example, you could write $k = i + j$, where k, i, and j are plural types.
- SIMD control statement semantics are implemented.
- SIMD control flow also controls the active set (see below).

An important concept in SIMD programming is the concept of the active set. The active set is the set of PEs that is enabled at any given time during execution. This set is defined by conditional tests in the program. The size of the active set can be no larger than the physical size of the PE array.

Plural data operations in an MPL program apply uniformly to all active PEs. For more explanation of the active set and program control flow, refer to [57].

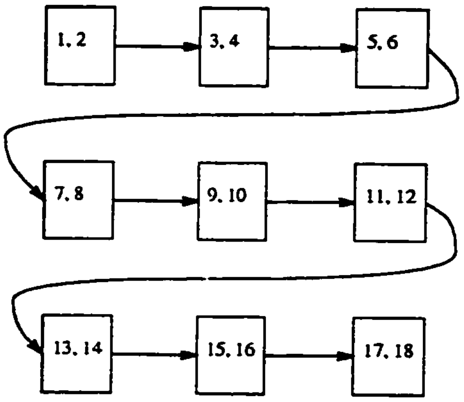
2.3.3 Input Key Characterization

As will be described later, it is important to correctly characterize the distribution of the data that is being sorted. It should be noted that there are two possible interpretations of the word "distribution". The first refers to the probability distribution of the values of the keys: value distribution (e.g. Are low-valued keys more common than high-valued keys?). The second interpretation refers to the way in which the keys are physically placed initially in the memory: allocation distribution (e.g. Are the keys already in sorted order? Are they in reverse sorted order?).

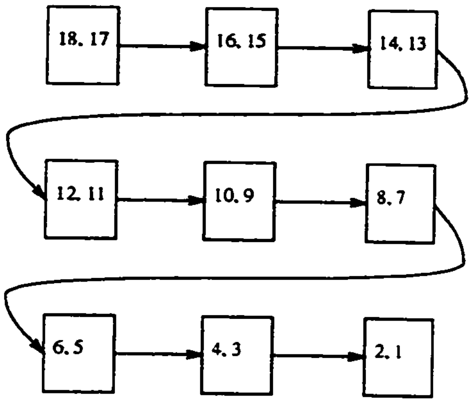
Thearling [73] proposes a test suite of inputs with which to evaluate the performance of sorting algorithms. We chose to test and evaluate the algorithms presented in this thesis according to a similar test suite.

One technique to characterize the value distribution of data is to measure the *entropy* of the data distribution. The Shannon entropy of a distribution [72] is defined as $\Sigma p_i|\log p_i|$ where p_i is the probability of symbol i occurring in the distribution. The entropy of the key distribution thus specifies the number of unique bits in the key. There are various techniques for producing keys whose individual bits are between 0 and 1 of entropy. We use the method proposed in [73]. The basic idea is to combine multiple keys having a uniform distribution into a single key by *AND*ing them together.

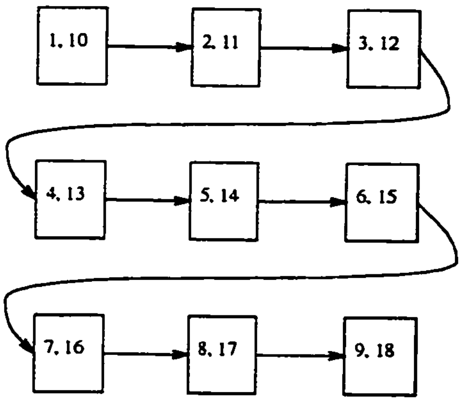
We define the *sparse* data distribution to have keys chosen uniformly at random from among 16 different keys. Following [75], we call the *Sparse/random* distribution to be the distribution that has 1% of keys chosen uniformly at random from the set of all 32-bit keys, and the remaining keys chosen from the sparse distribution defined above.



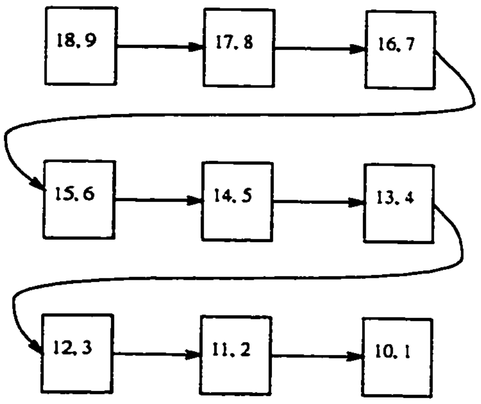
a) Presorted block order



b) Reverse block order



c) Presorted cyclic order



d) Reverse cyclic order

Figure 1: Example of different allocation distribution patterns

No. of Keys	Key Size	Value Distribution	Allocation Distribution
2^{10}	32 bits	Random	Random
2^4 2^5 2^6 2^7 2^8 2^9 2^{10} 2^{11}	32 bits	Random	Random
2^{10}	8 bits 16 bits 32 bits 64 bits	Random	Random

2^{10}	32 bits	Entropy = 0.811
		Entropy = 0.544
		Entropy = 0.337
		Entropy = 0.201
		Entropy = 0.0
		Sparse 16
		Sparse 16/Random
			Random

2^{10}	32 bits	Random	Presorted Block Order Presorted Cyclic Order Reverse Sorted Block Order Reverse sorted Cyclic Order

Table 1: Test cases for sorting evaluation on MasPar MP-1

There are several allocation patterns that are common, such as initially presorted and reverse sorted data. We included in our test suite, as in [73], four different allocation patterns: presorted block order, reverse block order, presorted cyclic order, and reverse cyclic order. The block layout requires neighboring elements of the sorted sequence to be a neighboring element in each processor's memory except for required breaks between processors, for any given fixed ordering of processors. The cyclic layout requires neighboring elements of the sorted sequence to be in the memories of neighboring processors. Figure 1 shows an example for each allocation pattern we used in this thesis for testing the sorting algorithms.

In Section 3.5 we evaluate and compare the performances of the algorithms along four different dimensions: number of keys, key size, distribution of key values and initial allocation of data to memory. Table 1 summarizes the test cases. The four parameters to be varied are listed across the top, and the base case is listed in the first row.

In addition to overall speed comparison there are many issues involved in evaluating and comparing sorting algorithms (see [73]) including : stability, determinism, memory efficiency, load balancing of the sorted data and the difficulties in extending the sort to perform a rank or to sort complex non-integer keys. We will discuss the above issues in Section 3.5.

2.4 Analysis of Algorithms on MasPar MP-1

This section defines an abstract model of the machine MasPar MP-1 that is used to describe and analyse all of the algorithms in this thesis. By describing a particular algorithm in terms of the abstract model, the analysis can be applied to other parallel machines and approximate run-times can be generated by substituting appropriate values for certain parameters.

Algorithm designers use a model of computation as a stable abstract machine without being concerned about developments in architectures. Any programming language implicitly defines an abstract machine. Thus, the programming MPL language defines an abstract model for MasPar MP-1.

We consider five main types of primitive operations provided by the Maspar MP-1. These are basic local primitive operations (this includes local operations such as assignments, comparisons, incrementing counters etc.), the *xnet* (X) or sending to nearest neighbor operation, the *scan* (S) or parallel prefix operation, the *reduceAdd* operation (C), and finally the *route* (R) operation that enables sending packets to arbitrary destinations. The *reduceAdd* operation is a communication primitive that takes a *plural* variable A_i, $0 \leq i < p$, as input and returns the *singular* value $\sum_{i=0}^{p-1} A_i$.

Operation	Time (in μ s)	Description of the operations
A	6	Time for an arithmetic operation
X	10	Distance-one nearest neighbor send time
R	1125	Average time to route to random destination
S	454	Time for *scan* operation
C	312	Time for *reduceAdd* operation

Table 2: Time for different operations on Maspar MP-1

Our performance analysis is in terms of these primitive operations. The costs of the primitive operations described in Table 2 are machine-dependent and are determined empirically.

We will model the runtime of an algorithm *algo*, T_{algo}, on a SIMD parallel computer using the sum of the computation time $T_{comp,algo}$ and the communication time $T_{comm,algo}$.

We predict the communication time of each algorithm based on the primitive operations *xnet* (X), *scan* (S), and *route* (R). The time complexity formula of the communication is

$$T_{comm,algo} = f(N,p)X + g(N,p)S + h(N,p)R$$

where f, g, h are functions that depend on the algorithm, and where N, the number of elements to be sorted, and p, the number of processors available, are the parameters of the algorithm.

Our model does not specify how local computation is to be modeled. We chose to do an asymptotic analysis of the local computation and then we assign a constant factor c_i to each term in the time complexity formula of the local computation,

$$T_{comp,algo} = c_1 A f_1(N,p) + c_2 A f_2(N,p) + ...$$

where the constant factors $c_i, i = 1, 2, ...$, were determined empirically, and A is nominally chosen to be the cost in μs of storing a constant into local memory (direct access) and incrementing a counter (assuming the operation is in a loop), that is in

MasPar MP-1 A is about 6 μs. It is useful to know that, on MasPar MP-1, the indirect memory accesses cost approximately 3 times more than direct accesses.

Chapter 3

A New Sorting Algorithm

3.1 Definitions

The input to our problem is a set X of N elements distributed over p processors of a parallel machine. Each processor i $(0 \leq i < p)$ is assumed initially to contain a unique subset of $\lfloor N/p \rfloor$ or $\lceil N/p \rceil$ of these elements in the array $Key_i[j]$ $(0 \leq j < N/p)$.

The $\sqrt{p} \times \sqrt{p}$ torus-connected array of processors (or two-dimensional torus) contains p processors arranged in a two-dimensional grid with wrap-around edges; each processor connected directly to its eight nearest neighbors (see Figure 2). More precisely, it corresponds to the undirected graph, $G = (V, E)$, with $V = \{(x, y) \mid x, y \in \langle\sqrt{p}\rangle\}$ and $E = \{((x, y), (x, y + 1 \; mod \; \sqrt{p})) \mid x \in \langle\sqrt{p}\rangle, y \in \langle\sqrt{p}\rangle\} \cup \{((x, y), (x + 1 \; mod \; \sqrt{p}, y)) \mid x \in \langle\sqrt{p}\rangle, y \in \langle\sqrt{p}\rangle\} \cup \{((x, y), (x + 1 \; mod \; \sqrt{p}, y + 1 \; mod \; \sqrt{p})) \mid x \in \langle\sqrt{p}\rangle, y \in \langle\sqrt{p}\rangle\} \cup \{((x, y), (x+1 \; mod \; \sqrt{p}, \sqrt{p}+y-1 \; mod \; \sqrt{p})) \mid x \in \langle\sqrt{p}\rangle, y \in \langle\sqrt{p}\rangle\}$, where $\langle n\rangle = \{0, \ldots, n-1\}$. Each link is considered to be bidirectional.

Let $P_{i,j}$ be the processor located in row i and column j. The processor $P_{0,0}$ lies in the upper left corner of the torus. An indexing scheme is a bijection from $\langle\sqrt{p}\rangle \times \langle\sqrt{p}\rangle$ to $\langle p\rangle$. The most natural indexing is the row-major order under which $P_{i,j}$ has index $i.\sqrt{p} + j$. The sorting problem on the 2-D torus is: Given a set X of N elements, stored with $\lfloor N/p \rfloor$ or $\lceil N/p \rceil$ elements per processor, and an indexing scheme for the processors, rearrange the elements so that every element in the processor indexed i is

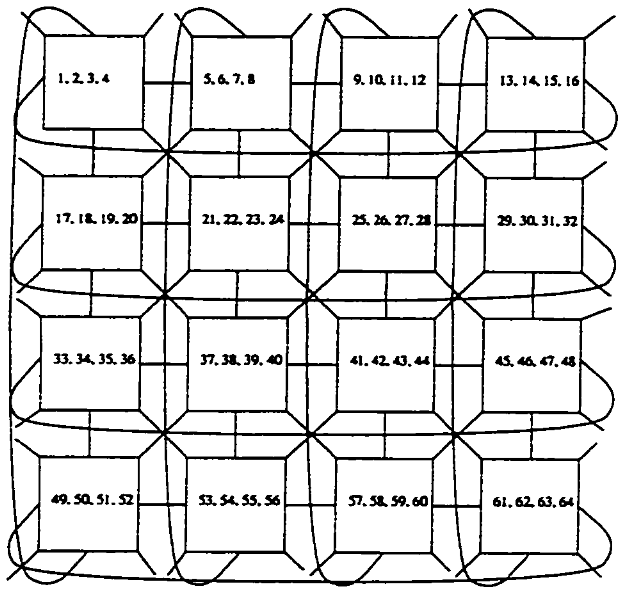


Figure 2: 4x4 torus-connected array of processors. The elements on the processors are sorted in row-major order. The diagonal wrap-around edges are not shown completely here.

less than or equal to every element in the processor indexed j whenever $0 \leq i < j < p$. In addition, we require the elements within any particular processor to be sorted. Sometimes, there is also a requirement that every processor have either $\lfloor N/p \rfloor$ or $\lceil N/p \rceil$ elements in the final sorted order; such an output is said to be a *balanced* output. Figure 2 gives an example of elements sorted in row-major order: note that the output in this example is balanced.

We use the term "with high probability" to mean with probability greater than $1 - 1/N^c$ for some constant $c \geq 1$. In our analysis, we make use of the following Chernoff bounds on the tails of the binomial distribution [5, 16].

Fact 1 *(Bernstein-Chernoff bounds) Let $S_{N,p}$ be a random variable having binomial distribution with parameters N and p ($S_{N,p}$ is the sum of N independent Bernoulli variables each with mean p). Then, for any h such that $0 \leq h \leq 1.8Np$,*

$$P(S_{N,p} \geq Np + h) \leq \exp\left(-h^2/3Np\right).$$

For any $h > 0$,

$$P(S_{N,p} \leq Np - h) \leq \exp\left(-h^2/2Np\right).$$

3.2 Description

In this section, we describe our algorithm for sorting, along with pseudocode for our implementation on the Maspar. The basic idea of the algorithm is to determine, for each input element x, the number of elements less than x, with an error less than M, where M will be specified later. This information can be used to send the element x directly into its intermediate position in the output array. A sorting algorithm with time complexity $O(M)$ can then be used to complete the sorting.

Phase 1: Find an approximate rank for each key, such that the calculated rank is within M of its actual rank among the keys.

Phase 2: Send the keys to their destination processors based on the approximate ranks.

Phase 3: Sort the keys locally within each processor.

Phase 4: Complete the sorting using odd-even transposition sort.

Figure 3: High-level Algorithm Description

A-ranksort consists of four basic phases which are summarized in Figure 3.

3.2.1 Phase 1: Counting Ranks

The heart of the algorithm is in the techniques used to compute approximate ranks. The objective is to divide the keys into small *segments*. A segment is said to be small if it contains less than M elements, where the value of M will be specified later. Initially all the keys are in the same (large) segment. The algorithm proceeds in rounds or iterations; each iteration divides the big segments into smaller segments. To divide a segment into smaller ones, we use a variable called *Block* associated with every key. Each possible value of *Block* defines a *bucket*. Further, all the keys in the same large bucket in iteration i (that is, the keys with the same *Block* value) will be in the same segment in iteration $i+1$. The variable *Block* associated with a key is initialized to be the most significant r bits of the key (the value of r will be specified later).

1. Initialize *Block* variables and all keys to be unranked.
2. While unranked keys remain do
    (a) Compute local ranks of each value in the Block array.
    (b) Perform a parallel prefix operation to find global ranks of keys.
    (c) For each small bucket, fix an arbitrary order for the keys in it. These keys are said to have been ranked, and will not participate any more in the ranking procedure.
    (d) Assign new segment numbers to elements in big buckets, and assign new block values by adding on the next few bits of the key.

Figure 4: High-level description of COMPUTE-RANKS procedure.

Keys that are in small buckets during iteration i will not participate in iteration $i+1$; we assign keys within the same small bucket an arbitrary order, which fixes the required approximate ranks. Thus, only the keys that are in the large buckets are as yet unranked and will participate in the next round. These keys are now assigned a new *Block* value. To do this, first the large buckets are all ranked, in such a way that the keys in bucket j have smaller ranks than the keys in bucket $j+1$. Clearly, if there are k large buckets, the bucket number can be represented using $s = \lceil \log k \rceil$ bits. If a key belongs to the large bucket ranked j in the current round, then its segment number in the next round is j. Then the new *Block* value of an unranked key is set to be the new segment number concatenated with the next $r - s$ unprocessed bits of the key (going from most significant bits to less significant bits). We repeat this process over the unranked keys until all the keys are ranked. This process is summarized in Figure 4.

We now describe our algorithm in greater detail. For clarity, we also present pseudocode for the procedure *Compute-Ranks* in Figure 5. Initially, all the keys belong to the same segment, the *Block* values are set to be equal to the r most significant bits of the keys, and all the keys are unranked.

For all *Block* values, lines 7-11 determine, for the unranked keys, how many times each value k appears in each processor; this value is stored in $Index_i[k]$, i.e.

```
COMPUTE-RANK($r$,$b$,$Key$)
 1 $nseg \leftarrow 1$
 2 $Offset[0] \leftarrow s \leftarrow 0$
 3 for $j \leftarrow 0$ to $N/p - 1$
 4     $Block_i[j] \leftarrow Key_i[j]\langle b-r, \ldots, b-1 \rangle$
 5     $Ranked_i[j] \leftarrow false$
 6 repeat
         /* compute local ranks */
 7     for $j \leftarrow 0$ to $2^r - 1$
 8         $Index_i[j] \leftarrow 0$
 9     for $j \leftarrow 0$ to $N/p - 1$
10         if $Ranked_i[j] = false$ then
11             increment $Index_i[Block_i[j]]$
         /* Find global ranks of elements */
12     $nlargebuckets \leftarrow 0$
13     for $j \leftarrow 0$ to $nseg - 1$
14         for $k \leftarrow j2^{r-s}$ to $(j+1)2^{r-s} - 1$
15             $count \leftarrow$ SUM($Index_i[k]$)
16             $Index_i[k] \leftarrow$SCAN($Index_i[k]$) + $Offset[j]$
17             if $count > M$ and $b > r$ then
18                 $Offset'[nlargebuckets] \leftarrow Offset[j]$
19                 $Bucket_i[k] \leftarrow nlargebuckets$
20                 increment $nlargebuckets$
21                 $SmallBucket_i[k] \leftarrow false$
22             else
23                 $SmallBucket_i[k] \leftarrow true$
24             $Offset[j] \leftarrow Offset[j] + count$
         /* Rank elements in small buckets */
25     for $j \leftarrow 0$ to $N/p - 1$
26         if $Ranked_i[j] = false$ and $SmallBucket_i[Block_i[j]] = true$ then
27             decrement $Index_i[Block_i[j]]$
28             $Rank_i[j] \leftarrow Index_i[Block_i[j]]$
29             $Ranked_i[j] \leftarrow true$
         /* Assign new Block values */
30     if $nlargebuckets > 0$ then
31         $s \leftarrow \lceil \log_2(nlargebuckets) \rceil$
32         $nseg \leftarrow nlargebuckets$
33         $b \leftarrow b - r + s$
34         if $b < r$ then $r \leftarrow b$
35         for $j \leftarrow 0$ to $N/p - 1$
36             if $Ranked_i[j] = false$
37                 $Block_i[j] \leftarrow Bucket_i[Block_i[j]] \ll (r-s) + Key_i[j]\langle b-r, \ldots, b-s-1 \rangle$
38         SWAP($Offset$,$Offset'$)
39 until $nlargebuckets = 0$
40 return $Rank_i$
```

Figure 5: Pseudo-code for the procedure Compute-Ranks

$$Index_i[k] = |\{j | 0 \leq j < N/p \ \wedge \ \neg Ranked_i[j] \ \wedge \ Block_i[j] = k\}|$$

Lines 12-24 iterate over each of the $nseg.2^{r-s}$ possible *Block* values, where *nseg* is the number of segments to process. All the keys in segment j have the possible *Block* values $k = j2^{r-s} \ldots (j+1)2^{r-s} - 1$, where the s most significant bits of the *Block* value k is equal to j. For each value of k, the algorithm uses a *Scan* operation to generate the array *Index*. This would compute the number of keys with Block value k that lie before the current processor in the indexing scheme. Clearly, these prefix operations alone do not calculate the exact ranks of keys. We need to add in the keys that are smaller than any keys in segment j as well as the keys in the same segment j but with smaller *Block* values. This offset is maintained in the *Offset* array. Thus, the final value of $Index_i[k]$ is defined as:

$$\begin{aligned} Index_i[k] \ = \ & |\{(m,n) | (0 \leq m \leq i \ \wedge \ 0 \leq n < N/p \ \wedge \ \neg Ranked_m[n] \ \wedge \\ & Block_m[n] = k) \ \vee \ (0 \leq m < p \ \wedge \ 0 \leq n < N/p \ \wedge \ \neg Ranked_m[n] \ \wedge \\ & Block_m[n] < k\}| \end{aligned}$$

While iterating through Lines 13-24, if the number of keys with *Block* value equal to k is more than M, i.e. $count > M$, then bucket k is considered to be a large bucket and will survive as a segment for the next round.

Having computed $Index_i[k]$, the rank of the last key in processor i with *Block* value equal to k, lines 25-29 fix an approximate rank to all the keys in small buckets. The ranks of the keys within a small bucket do not matter, since the keys in are assigned ranks that are within M of their actual rank.

Lines 31-37 assign a new *Block* value for all the unranked keys. The *Block* values are considered to be bit-strings $\langle \alpha_{r-1} \ldots \alpha_0 \rangle$, where $\alpha_0, \ldots, \alpha_{r-1} \in \{0,1\}$. The s most significant bits represent the new segment number and the $(r-s)$ least significant bits are the next unprocessed $(r-s)$ bits from the keys, moving from the most significant to least significant bits.

The algorithm will eventually rank all the keys, so long as at least one new bit of the keys is processed in every iteration. The progress condition of the algorithm is that the number of large buckets found in one iteration is at most 2^{r-1}. It is not hard to see that this is always *true* if $M \geq \lfloor \frac{N}{2^{r-1}+1} \rfloor$. Thus any value of $M \geq N/2^{r-t}$ where $t \geq 1$ will suffice to guarantee termination of the loop.

3.2.2 Phase 2: Routing Keys to Approximate Destinations

For each key, based on its approximate rank found in Phase 1, we derive its destination processor. Then, we send the keys to their respective processors.

If there are 2^j processors and 2^k keys per processor, the low order k bits specify the array index. The next j bits then specify the processor index. If the number of keys per processor is not a power of 2, then an appropriate divide operation suffices to find the destination.

When a key is sent to its destination processor, there is no need to append the array index of the destination to the key message packet, as required in radix sort [15, 75]. This is because the keys are not necessarily ranked within segments. This observation reduces the time required for routing keys.

3.2.3 Phase 3: Local Sorting

The third phase sorts the keys locally within each processor. We use the standard serial radix sort in which each pass is implemented using a counting sort [20]. Radix sort was used in our implementation, since it is significantly faster than comparison sorts such as quicksort.

3.2.4 Phase 4: Final Sorting

In Phase 4 the algorithm performs at most $\lceil M/(N/p) \rceil + 1$ steps of odd-even transposition sort on the overall *Key* array in row-major order. Each step of odd-even transposition sort consists of a merge/split operation on two sorted lists located in two consecutive processors.

3.3 Performance Analysis

In this section, we will prove average-case and worst-case bounds on the performance of *A-ranksort*. We will prove bounds on the time required for each phase of the overall algorithm described in Figure 3. We analyse the algorithm in terms of the basic operation costs described in Table 2.

3.3.1 Average-case Analysis

Lemma 1 *Each iteration in Phase 1 takes time at most* $c_1 A 2^r + c_2 A(N/p) + 2^r S$ *time where* c_1 *and* c_2 *are constants.*

Proof: We use Figure 5 in the analysis of Phase 1. The initializations in lines 1 - 6 take at most $O(A(N/p))$ time. It is easy to see that computing local ranks (lines 7-11) takes at most $O(2^r + (N/p))$ local operations. Computing global ranks (lines 8-24) uses an additional $O(2^r)$ local arithmetic operations and 2^r scan operations. Further computing ranks of elements in small buckets requires at most $O(A(N/p))$ time. As we will see below, lines 30-38 are not executed in the average-case. This proves the lemma.

□

Lemma 2 *Phase 2 can be completed in* $(N/p)R$ *time.*

Proof: Obvious. □

Lemma 3 *Phase 3 can be completed in*

$$(b/q)(c_1 A 2^q + c_2 A(N/p))$$

where q *is the size of the radix used in the local radix sort.*

Proof: The analysis of Radix sort is omitted here. The interested reader is referred to [20]. □

Lemma 4 *Phase 4 can be completed in $M(c_1X + c_2A)$ time, where c_1 and c_2 are constants.*

Proof: At the end of Phase 3, each element is at most M ranks away from its real rank. Therefore, M steps of odd-even transposition sort on the entire sequence suffice to sort the input. (The description of odd-even transposition sort and the above claim of its behavior are omitted here. Details can be found in [55]).

□

All that is required is to determine the number of iterations taken by the loop in the procedure *Count-Ranks*. A simple observation is that if the input elements are chosen uniformly at random from the set $\{0, \ldots, 2^b - 1\}$, then each combination of the first r bits appears approximately the same number of times among the input elements. In other words, the elements can be split into approximately equal sized buckets on the basis of the first r bits. We formalize the above observation in the following lemma.

Lemma 5 *For randomly chosen input, the loop in the* COMPUTE-RANKS *procedure described in Figure 4 executes once with high probability.*

Proof: From the description in Section 3.2.1 it is clear that the loop executes as long as there are buckets that contain more than M keys. We prove now that if the input is randomly chosen, then in the first iteration, all buckets are small with high probability. Therefore, all keys will be ranked in the first iteration, and the loop will exit after the first iteration.

There are 2^r possible buckets. If the input is chosen uniformly at random from the set $\{0, 2^b - 1\}$, then the probability that a key has *Block* value equal to k where $0 \leq k \leq 2^r - 1$ is $1/2^r$. Therefore, the expected number of keys with *Block* value k is $N/2^r$. Using Fact 1, and setting $M = N/2^{r-1}$ we obtain:

$$Prob(\ \#\text{ of keys with } Block \text{ value } k > N/2^{r-1}) \leq e^{\frac{-N^2/2^{r+1}}{3N/2^r}} \leq e^{-N/6}$$

Thus, with very high probability, the size of the bucket with *Block* value equal to k is at most M. Since there are 2^r buckets, the probability that there is a k such that there are greater than M keys with *Block* value equal to k is less than $2^r e^{-N/6} \leq N^{-c}$ as long as r is $o(N - c\log N)$.

□

Choosing $2^r = p/4$ and $M = N/2^{r-1}$, and combining Lemmas 1 to 5, we get the following:

Theorem 1 *Given N elements chosen uniformly at random from the set $\{0, 2^b - 1\}$, the algorithm A-ranksort completes sorting the inputs in time $(c_1(N/p)+pc_2+bc_3)A+(p/4)S + (R + c_4X)(N/p)$ with high probability, where c_1, c_2, c_3, c_4 are constants.*

3.3.2 Worst-case Analysis

Theorem 2 *Given any N elements placed at a p processor mesh, the algorithm* A-ranksort *completes in time at most $((b-r)(c_1(N/p)+pc_2)+bc_3)A+(b-r)(p/4)S+(R+c_5X)(N/p)$ where c_1, c_2, c_3, c_4, c_5 are constants.*

Proof: Follows from the fact that there are at most $b - r$ iterations in the worst case, and from Lemmas 1 to 4. □

3.3.3 Analysis for Other Input Distributions

Lemma 5 shows that when the input is uniformly distributed, all buckets are small with high probability. When we consider other input distributions, this may not be the case. For example, for a sparse distribution of the type defined in Section 2.3.3, as many as 16 buckets could be large at the end of any iteration. However, in this case, in each iteration at least $r - 4$ new bits would be processed and thus a maximum of $(b - r)/(r - 4) + 1 = (b - 4)/(r - 4)$ iterations would suffice to complete the ranking procedure; it would be interesting to find the expected number of iterations in COMPUTE-RANKS procedure instead.

It would be useful to prove performance bounds for the algorithm for fixed values of M and r under an input distribution with entropy H, where $0 \leq H \leq b$. While we were unable to prove such bounds, empirical observations show that regardless of the entropy of the distribution, the number of iterations is usually relatively small. It would be interesting to prove that the number of iterations in COMPUTE-RANKS procedure is bounded by a small constant with high probability.

It is important to notice that whenever the entropy is high (close to b), the number of keys in the largest bucket is closer to $M = N/2^{r-1}$, thus it takes only few more new bits to process in order to divide this large bucket into small buckets. And whenever the entropy is low (close to 0) the number of large buckets is low, thus the number of new bits to process in the next iteration is closer to r; in this case the number of iterations is bounded by b/r.

3.3.4 Space Requirements

We briefly describe here the space requirements of our algorithm. In Phase 1, an r bit array of size N/P is required to store the *Block* variables, and an array of size 2^r is required to calculate ranks. Space can be reused for all other variables. In Phase 2, keys will be stored at both origins and destinations of routes, thus doubling the space required for the keys alone. In Phase 3, similarly, we need additional space for storing the sorted keys. We also require a 2^q size array to do the local radix sort. Phase 4 can be done with an extra key-size array by a clever implementation of the merge-split operation. Therefore the total space requirements of the algorithm can be summarized as follows:

Type of variable	Array size (per processor)
Key type	N/p
$\log N$ bits	$2^r + 2^q$

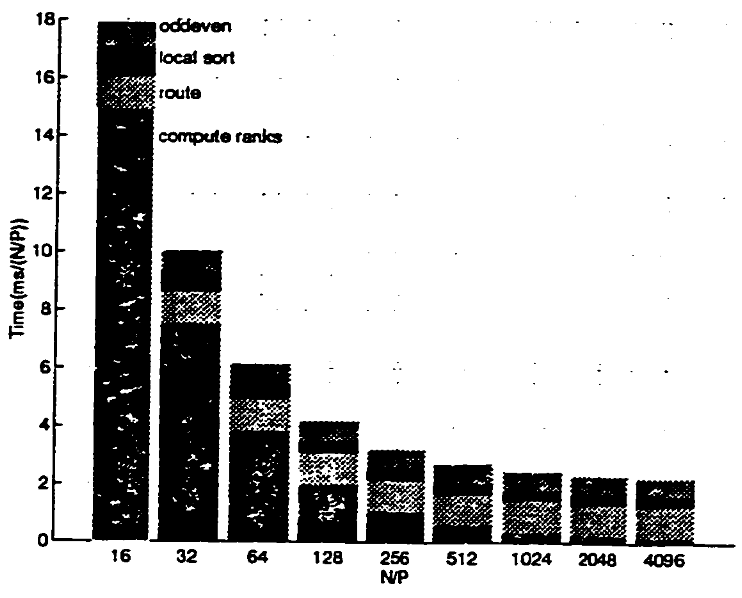


Figure 6: Performance of *A-ranksort*

r bits N/p

3.4 Empirical Results

All our results and graphs are for 32-bit keys on uniformly distributed data, unless specified otherwise. Figure 6 shows the performance of our algorithm on different values of N/p. By examining our program *A-ranksort*, and using experimental results, we derived the constants referred to in the analysis in Section 3.3. The costs of primitive operations are shown in Table 2.

We can predict the performance of the algorithm using the following formula:

$$T_{A-ranksort} = T_{computeranks} + T_{route} + T_{localsort} + T_{oddeven}$$

where

$$T_{computeranks} = A2^r + 22.5A(N/p) + 2^r S$$

$$T_{route} = (N/p)R$$

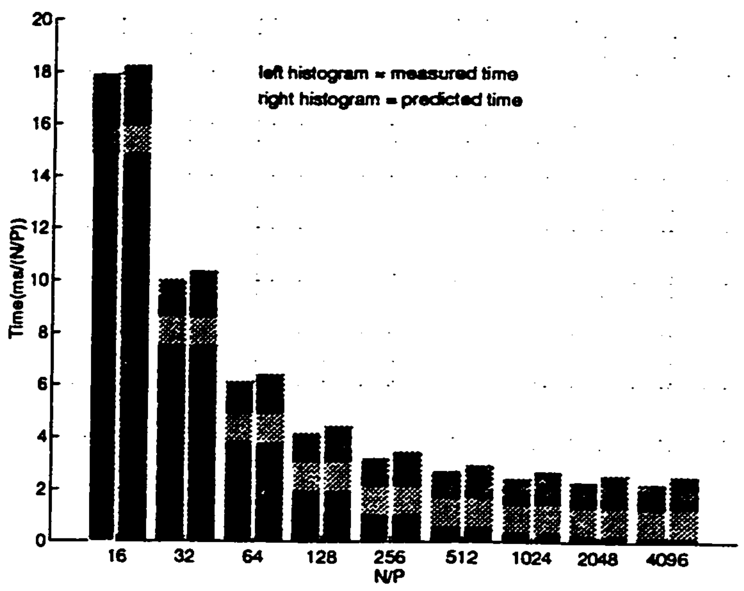


Figure 7: Predicted versus measured times for *A-ranksort*

$$T_{localsort} = (b/q)(2.7A2^q + 13.3A(N/p))$$

$$T_{oddeven} = 17AM + 0.75MX$$

Figure 7 shows the closeness of the measured times to the time predicted by our formula.

The choice of r and M clearly affects the performance of the algorithm, as does the input distribution. In particular, as the value of r increases, the time taken by Phase 1 increases, but smaller values of r imply larger values of M which in turn, increases the cost of the final sorting. Figure 8 shows the effect of varying r on the performance of the algorithm. We note here that to get an improved version of *A-ranksort*, we could calculate the best values of r for different values of N/p and choose the best r for the given input. For example, the graph suggests that for $N/p \leq 128$, the value of r that would minimize the running time would be 7 and similarly for $N/p > 128$, the value $r = 9$ could be the best choice of r.

Thearling [73] proposes a number of benchmarks to study the performance of sorting algorithms on different types of input distributions, key sizes, numbers of keys, and different allocation distributions. Figure 9 shows the performance of our algorithm on input distributions with varying entropies. Our algorithm performs

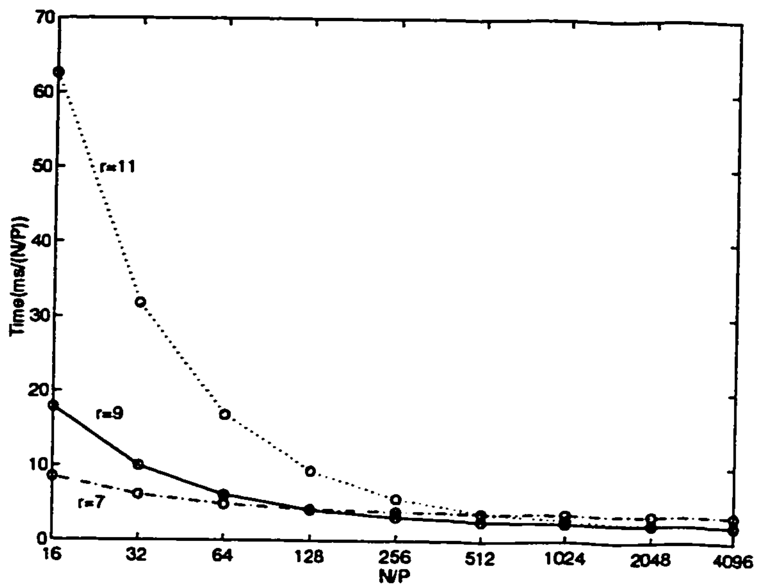


Figure 8: Performance of *A-ranksort* for different choices of r

consistently well on all these distributions. Figure 10 demonstrates the performance of our algorithm for different key sizes: the algorithm appears to scale well for higher key sizes.

We tested our algorithm on different kinds of presorted inputs. It can be seen from Figure 11 that it performs well on block sorted, and reverse sorted orders, but its performance on the routing step degrades considerably on cyclic orders. This is easily understood by noting that on an input in cyclic sorted order, each routing step would require routing an element from every processor to the same processor thus causing congestion and delays. This can be avoided by randomly permuting the input before or after computing the ranks and then performing the routing step.

3.5 Comparison with Existing Sorting Algorithms

There have been many efforts to find the best algorithm for sorting on parallel machines; Blelloch *et al.* [15] provided a quantitative analysis of the circumstances under which different algorithms for sorting on the CM-2 were advantageous, and a similar effort was made by Hightower *et al.* [40] on Maspar's MP-1, by Dickmann *et al.* for

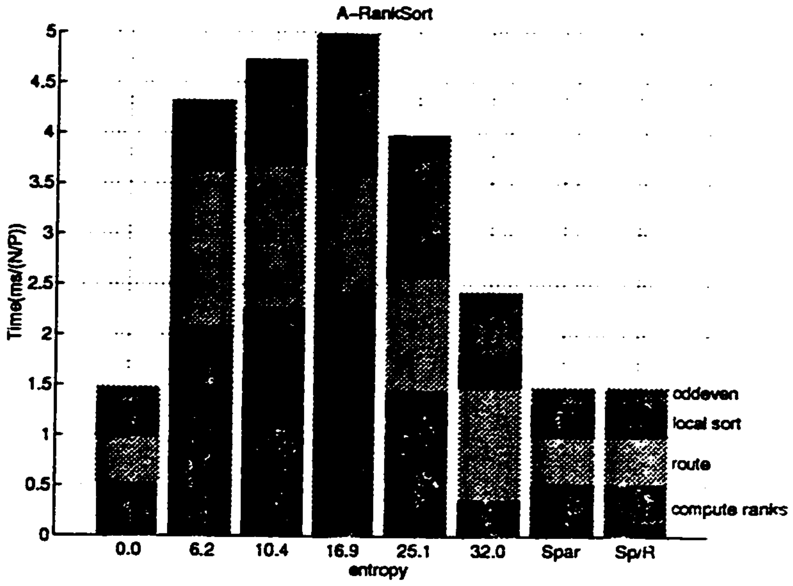


Figure 9: Performance of *A-ranksort* on different input distributions

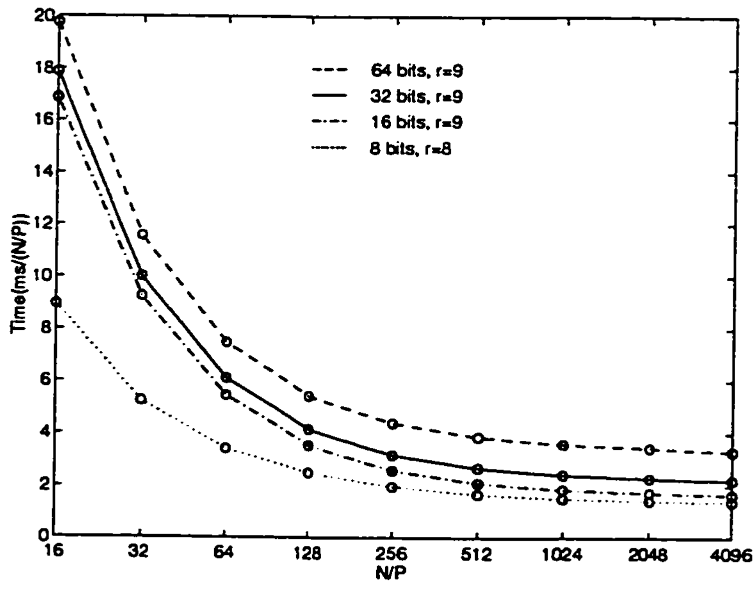


Figure 10: Performance of *A-ranksort* for different key sizes

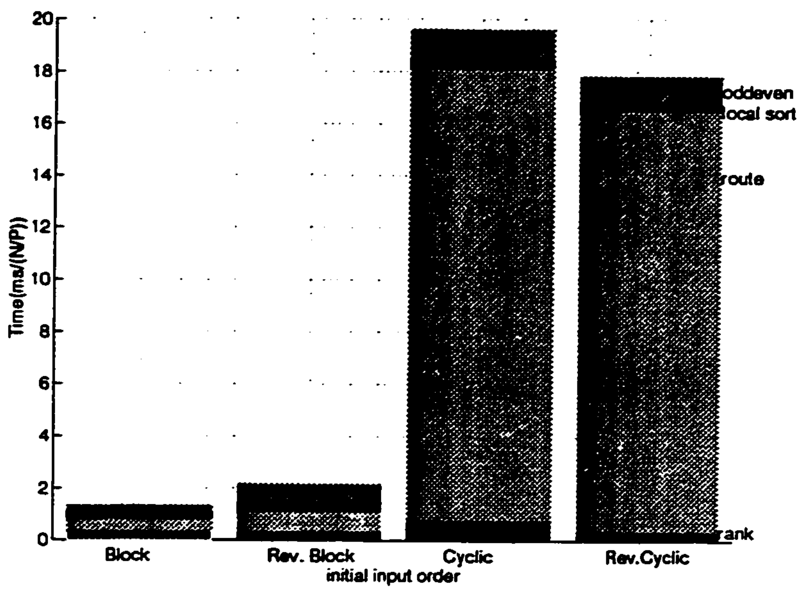


Figure 11: Performance of *A-ranksort* on different initial input orders

a Parsytec GCel [24], and by Helman *et al.* on a number of platforms [38]. In order to evaluate the relative performance of our algorithm, we selected the three most promising alternatives for comparison on MasPar MP-1: bitonic sort [63, 6, 23], radix sort [15, 23, 45], and B-flashsort [40].

Figure 12 shows the relative performance for a uniformly distributed data of our algorithm with *Radix sort, B-flashsort, Bitonic sort* and *B-flashsort* with load balancing. The performance of *A-ranksort* is slightly better than the performance of *B-flashsort* when load balancing is required by the application, but it is quite a bit worse than *B-flashsort* when load balancing is not required.

Figure 13 compares the performance of the *B-flashsort* (without load balancing), *Radix-sort*, and *A-ranksort* for a variety of data distributions. We can see that for certain data distributions, specifically, for the sparse ones, *B-flashsort* fails to sort, owing to space limitations. Even on distributions with smaller entropies, its performance degrades considerably owing to load imbalance. The performance of *A-ranksort* is also seen to be the best among all the algorithms for all data distributions, thus making *A-ranksort* the algorithm of choice for many practical applications when the distribution of data is not uniformly random.

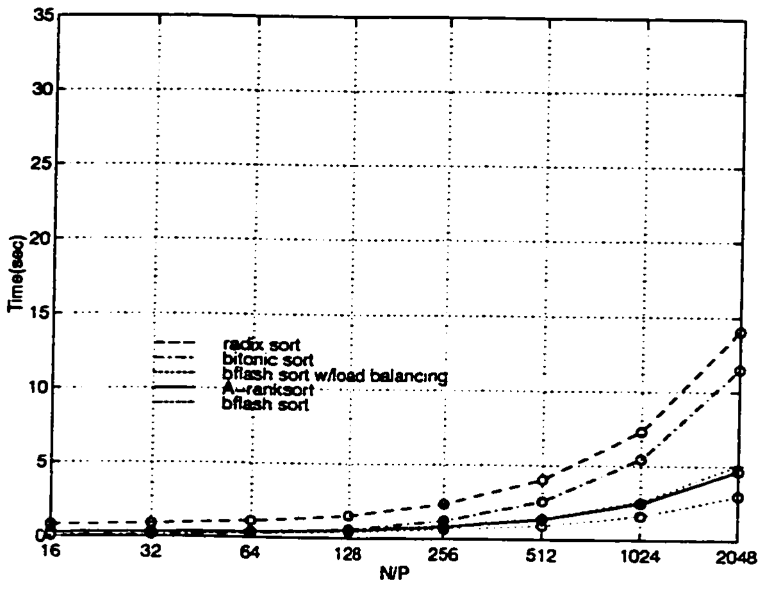


Figure 12: Comparison of running times for sorting 32-bit keys that are uniformly distributed

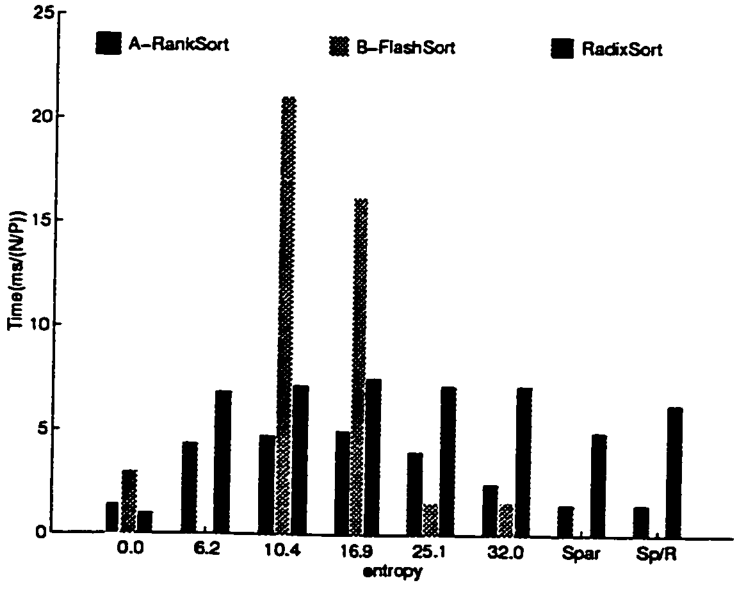


Figure 13: Comparison of algorithms on different data distributions, key size = 32-bits

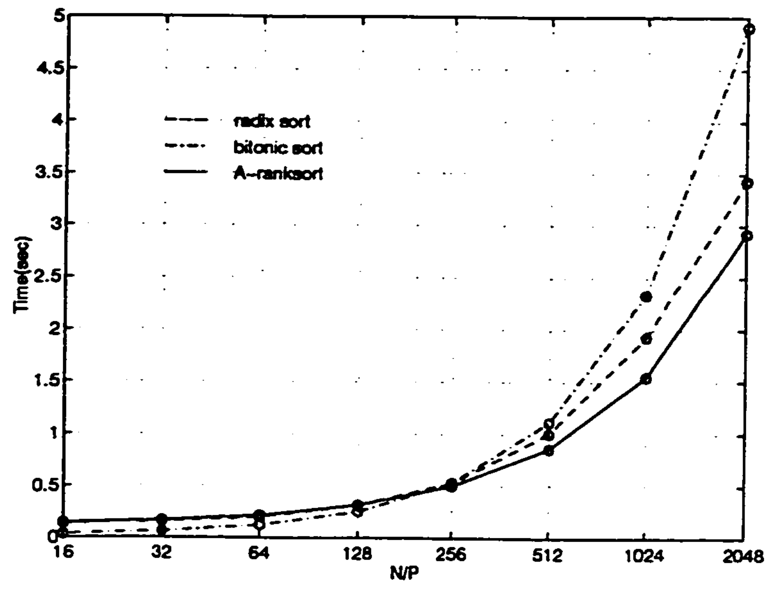


Figure 14: Comparison of running times for sorting 8-bit keys that are uniformly distributed

We have explained above that our algorithm does not perform well on all initial input orders. It should be noted that there are initial orders for which *Radix sort* has very bad performance as well [23], though it performs well on all the distributions in our test suite. Similarly *B-flashsort* appears to be indifferent to the initial order of the input.

As shown in Figures 14, 15, 16, 17, all sorting algorithms are dependent on the size of their key since the cost of local computation (e.g counting, comparison, memory access) increases, and the total amount of data (in bits) that must be permuted increases. In case of radix sort, the key size also affects the number of permutations that must be performed and the size of the histogram used in counting.

Figure 18 shows the running times of sorting 1024 keys per processor for different key sizes. For 16-bit keys, *B-flashsort* is approximately 2 times faster than *A-ranksort*, for 32-bit keys it is 1.6 times faster, and for 64-bit keys it is only 1.2 times faster. As the size of the keys increases the performance of *A-ranksort* degrades more slowly than the other algorithms, this is indicated by the graphs.

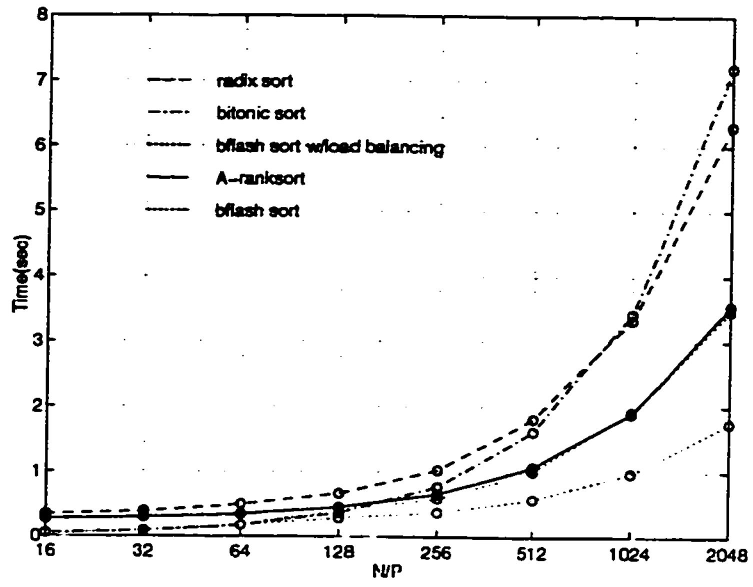


Figure 15: Comparison of running times for sorting 16-bit keys that are uniformly distributed

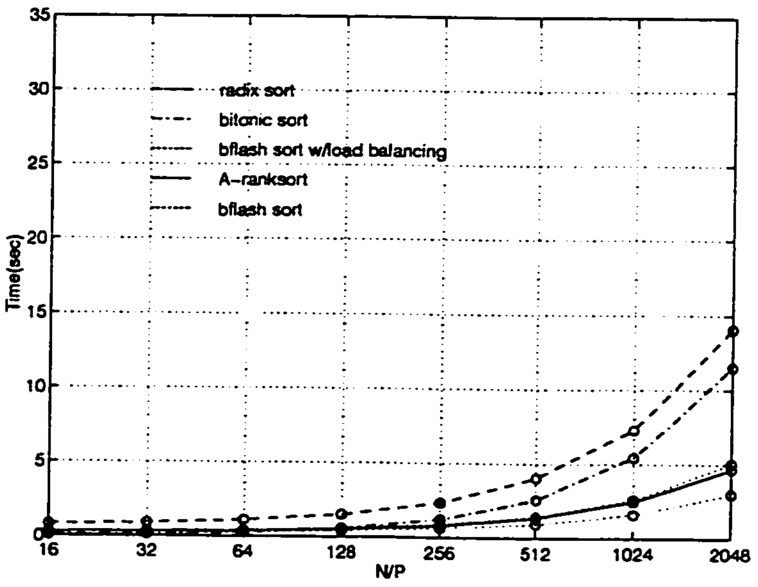


Figure 16: Comparison of running times for sorting 32-bit keys that are uniformly distributed

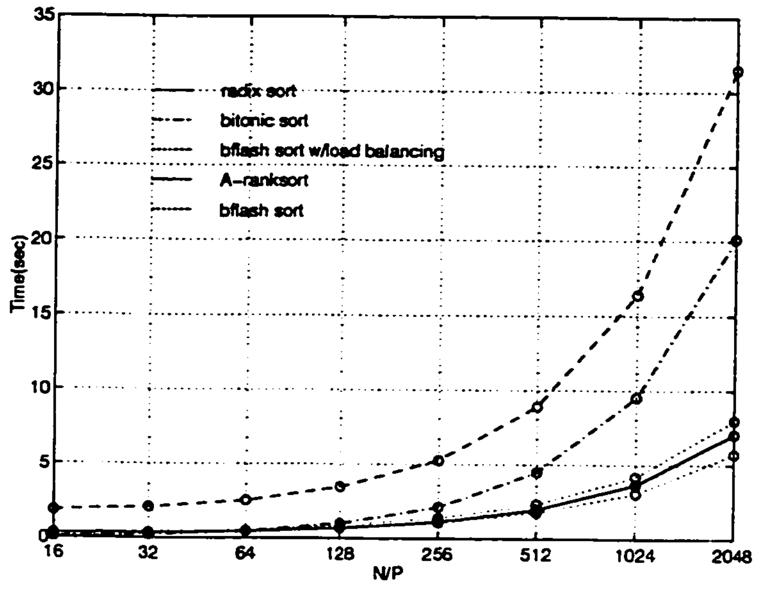


Figure 17: Comparison of running times for sorting 64-bit keys that are uniformly distributed

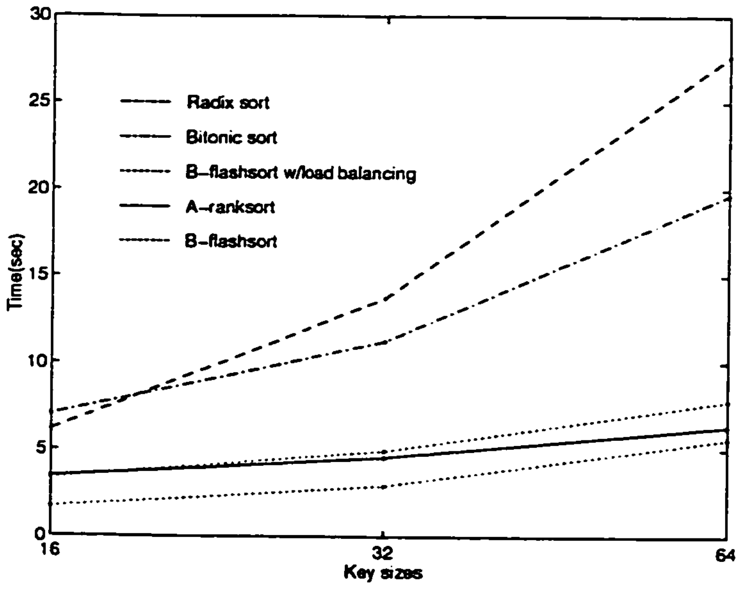


Figure 18: Comparison of the running times of sorting uniformly distributed data for different key sizes (N/p=2048)

In addition to overall speed comparison there are many issues involved in evaluating and comparing sorting algorithms, including : stability, determinism, load balancing of the sorted data, memory efficiency, and the difficulties in extending the sort to perform a rank or to sort complex non-integer keys (see [73]).

3.5.1 Stability

A sorting algorithm is *stable* if numbers with the same value appear in the output array in the same order as they do in the input array. The property of stability is important only when the elements to be sorted consists of a key and a satellite data that is carried around with the key being sorted. *Radix sort* is stable, while the other three sorts are not. *A-ranksort* can be made stable by appending the array index of the destination to the key message packet, as required in *Radix sort* (see Section 3.2.2); this will slow down the routing phase by a factor of 1.2. *Bitonic sort* and *B-flashsort* can be made stable by appending the original position of the data to the least significant bits of the key. In this case, however, not only must the tag be carried around during the *route* operations, it must also be used in the comparisons.

3.5.2 Determinism

An algorithm is said to be deterministic if its behavior (including running time) on the same input data is always identical. *Bitonic sort* is oblivious to the data or its initial allocation in memory and will always take the same amount of time. However, the other three algorithms are dependent on the communication network to perform random or irregular permutations. Their running time could vary if the communication network is non-deterministic or if the initial allocation of the data to memory is changed causing different routing patterns which may or may not be more efficient for the given architecture. In addition, the performance of *B-flashsort* depends heavily on the random sampling of the splitting values. In case of poor sampling the performance of *B-flashsort* degrades heavily owing to load imbalance.

3.5.3 Data Balance

The distribution of N elements over p processors is said to be balanced, if each processor holds N/p elements. *B-flashsort* does result in unbalanced data allocations; the processors have different number of keys at the end of sorting. This imbalance increases due to poor sampling. Furthermore, this imbalance depends on the value distribution of the keys; for example, the imbalance increases with decreasing entropy. The cost of the local sort in *B-flashsort* increases proportionally to the imbalance. We can see in Figure 13 that for certain data distributions, specifically, for the sparse ones, *B-flashsort* fails to sort, owing to space limitations. *A-ranksort*, *Radix sort*, and *Bitonic sort* do not result in unbalanced data allocation. Although some applications of sorting do not require that the processors loads be exactly balanced, many do. Load balancing can be performed by first scanning to determine the destination of each sorted key and then routing the keys to their final destinations. The dominant cost in load balancing is the extra *route* operation. We implemented a version of *B-flashsort* with load balancing, the additional cost was up to 70 percent, and the algorithm is outperformed by *A-ranksort*, as shown in Figure 12.

3.5.4 Memory Efficiency

An important concern is the space required by each sorting algorithm. *Bitonic sort* executes in place and therefore requires only a small constant amount of additional memory. *A-ranksort* and *Radix sort* require an additional N/p key type variables per processor that are needed to store the keys after the *route* (the *route* cannot be executed in place). Also, 2^r 32-bit words of space per processor are needed for holding the bucket sums. *A-ranksort* requires an additional 2^q 32-bit words used for local *radix sort*, and N/p r-bit words for holding the *block* values. The space requirement of *B-flashsort* depends heavily on the data distribution and the choice of the splitters. For uniformly distributed data, the total space used is up to $2N/p$ key type variables per processor with high probability. Experiments show that for certain data distributions, specifically, for the sparse ones, *B-flashsort* fails to sort, owing to space limitations. Even on distributions with smaller entropies, the space requirement increases considerably owing to load imbalance (see the analysis of the expected space used by *B-flashsort* for uniformly distributed data in [40], it is perhaps interesting to do the same for other data distributions).

3.5.5 Rank

For a vector of keys, the *rank* operation returns to each key the rank it would attain if the vector were sorted. Often, in practice, ranking the keys is more useful than sorting. In many applications there is a large block of auxiliary information associated with each key. The *rank* operation allows the user to rank the keys and than send the data to the final sorted position. To implement a *rank* operation in terms of a sort, we carry around a return address (the key's original position) of each key during the sort. Once sorted, the enumeration of their positions is sent back, as their rank, to the location specified by the return address. By implementing the *rank* operation using *Radix sort*, we can avoid the additional *route* operation by omitting the last *route* operation, and sending the rank directly back to the position specified by the return address. Furthermore, as each block of the key is used by *Radix sort*, that block can be thrown away, thereby decreasing the cost of subsequent *route* operations. Because of this, the running time of the *rank* operation based on *Radix sort* is probabbly marginally more expensive than that of *Radix sort*. Implementing the *rank* operation using *A-ranksort*, *B -flashsort* and *Bitonic sort* slows down the algorithm by carrying the return address around and by using an additional *route* operation.

3.5.6 Complex Keys

There are many database sorting applications where the keys are made up of complex combinations of smaller fields. With comparison-based sorting such as *Bitonic* *sort* and *B-flashsort*, sorting such complex keys is relatively easy by defining a comparison function. For counting based algorithms such as *A-ranksort* and *Radix sort*, it might be difficult to convert the complex keys to integers and the resulting keys may be rather long. For random distributions, while the performance of *Radix sort* will degrade proportionally to the key size, *A-ranksort* will perform well with high probability as shown in Section 3.3.

Chapter 4

A New Selection Algorithm

The comparison-based model may not always be the most natural one for the study of sorting and selection algorithms, since real machines allow many other operations besides comparison. Many algorithms (e.g. *Radix sort*) were designed for sorting integers using other operations besides comparison (e.g. indirect address, left shift, right shift [2, 3, 52]). Integer sorting is in fact one of the sorting problems most frequently encountered. We believe that integer selection is an interesting problem as well.

This chapter presents a fast and deterministic algorithm, *A-Select*, for finding the element of rank s given a set of N integers distributed across a parallel machine, for a given integer s. Note that the median finding algorithm is a special case of the selection problem where s is equal to $\lceil N/2 \rceil$. The selection algorithm that we describe in this chapter uses the binary representation of the input elements, while all the other known algorithms for selection are based on comparing pairs of elements and they must make at least $N-1$ comparisons [45].

4.1 Description and Performance Analysis

In this section, we describe the algorithm *A-select*, along with pseudocode for our implementation on the MasPar. Our algorithm, as in *A-ranksort*, relies on the representation of keys as b-bit integers, and examines the input keys r bits at a time, starting with the most significant block of r bits in each key. The algorithm proceeds in rounds or iterations; each iteration divides the input keys into buckets; each possible value of the r bits defines a bucket. It is obvious that the keys in bucket i are less than the keys in bucket j, for all $i < j$.

Only the keys that are in the bucket k_i during iteration i, such that $\sum_{j=0}^{k_i-1} B_j < s_i$ and $\sum_{j=0}^{k_i} B_j \geq s_i$, will participate in the next iteration; where B_j is the number of keys in bucket j. In iteration $i+1$ we use the next r bits of each key and select the key of rank $s_{i+1} = s_i - \sum_{j=0}^{k_i-1} B_j$, where $s_1 = s$. After b/r iterations the keys in bucket $k_{b/r}$ are all equal and the result of the algorithm is one of those keys, i.e $\sum_{i=1}^{b/r} k_i 2^{b-ir}$.

We now present the pseudocode for *A-select* in Figure 19 and describe it in greater detail. The algorithm selects the sth element from the vectors $Key_i[0..n_i - 1]$, for $0 \leq i < p$. For all block values $(Key_i[j]\langle b-r, \ldots, b-1\rangle$, for $0 \leq j < n_i$ and $0 \leq i < p)$, lines 4-9 determine for each value k and in each processor how many keys appear in buckets $0 \ldots k$; this value is stored in $Histo_i[k]$.

Lines 10-17 select the bucket k, using the binary search technique, such that $\sum_{i=0}^{p-1} Histo_i[k-1] < s \leq \sum_{i=0}^{p-1} Histo_i[k]$. It is clear that the key with rank s belong to the bucket k, thus only the keys with block value k will participate in the next iteration, from which we select the key with rank $s - \sum_{i=0}^{p-1} Histo_i[k-1]$. We use the vector *count* stored in the ACU, instead of a singular variable, in order to avoid the use of an extra *reduceAdd* operation in line 26. It should be noticed that for any selected bucket k, it is always true that the entity $count[k-1]$ at line 26 is already computed at line 14, while searching for bucket k, and is equal to $\sum_{i=0}^{p-1} Histo_i[k-1]$.

It is not difficult to see that the binary representation of the value *res* of the sth element of the input keys is equal to $(k_1' k_2' \ldots k_{b/r}')$, where k_j is equal to the value

```
$A$-select($Key_i$,$n_i$,$s$,$b$,$r$)
res ← 0
while $b > 0$
do
    if $b < r$ then $r \leftarrow b$
    /* local histogramming */
    for $j = 0$ to $2^r - 1$ do $Histo_i[j] = 0$
    for $j = 0$ to $n_i - 1$
        do increment $Histo_i[Key_i[j]\langle b - r, \ldots, b - 1\rangle]$
    for $j = 1$ to $2^r - 1$
        do $Histo_i[j] = Histo_i[j] + Histo_i[j - 1]$
    /* select bucket k where the sth key belong, using binary search */
    $h \leftarrow 2^{r-1}$
    $k \leftarrow h - 1$
    while $h > 1$
    do $h \leftarrow h/2$
        $count[k] \leftarrow reduceAdd(Histo_i[k])$
        if $count[k] < s$ then $k \leftarrow k + h$
        else $k \leftarrow k - h$
    if $reduceAdd(Histo_i[k]) < s$ then increment $k$
    $res \leftarrow res + k \ll (b - r)$
    if $b > r$ then
        /* only keys in bucket $k$ will participate in the next iteration */
        $m_i \leftarrow 0$
        for $j = 0$ to $n_i - 1$
            do if $Key_i[j]\langle b - r, \ldots, b - 1\rangle = k$ then
                    $SWAP(Key_i[m_i], Key_i[j])$
                    increment $m_i$
        $n_i = m_i$
        /* select the key in bucket $k$ with rank $s = s - \sum_{j=0}^{k-1} B_j$ */
        if $k > 0$ then $s \leftarrow s - count[k - 1]$
    $b \leftarrow b - r$;
od
return res
```

Figure 19: Pseudo-code for *A-select*

of the selected bucket in iteration j and $k_j^{'}$ is the binary representation of k_j; i.e $(k_1^{'}k_2^{'}\dots k_{b/r}^{'}) = \sum_{i=1}^{b/r} k_i 2^{b-ir}$. Lines 20-25 determine the keys that will participate in the next iteration.

It is clear from the pseudocode that the worst-case time complexity of the local computation is $T_{comp,A-Select} = b/r(c_1 AN/p + c_2 A2^r)$, where c_1 and c_2 are determined empirically and they are machine dependent. To analyze the complexity of the communication time we observe that the *while* loop of lines 12-16 runs $r-1$ times, and an extra *reduceAdd* operation is used in line 17. Since the algorithm runs in b/r iterations, the total number of *reduceAdd* operations is b. It follows that the worst-case time complexity of *A-select* is

$$\begin{aligned} T_{A-Select} &= T_{comp,A-Select} + T_{comm,A-Select} \\ &= b/r(c_1 AN/p + c_2 A2^r) + bC \end{aligned}$$

The space requirement of *A-select* is a constant number of singular variables, the array of integers *count* of size 2^r stored in the ACU, and an array of integers $Histo_i$ of size 2^r in each processor.

4.2 Empirical Results

The choice of r clearly affects the performance of the local computation of the algorithm. In particular, as the value of r increases, the time taken by line 5 and lines 8-9 increases, and the number of iterations decreases. Also, the input distribution affects the load balance of the processors; the running time of lines 6-7 and lines 21-24 depends on how many keys in each processor will participate in the actual iteration. For uniformly distributed data the processors are load balanced in all iterations with high probability. As shown in the previous section, the number of *reduceAdd* operations is equal to b and do not depend on r nor on the input data distribution.

All our tests are done for $r = 8$. We note here that to get an improved version of *A-select*, we could calculate the best value of r using the worst-case time complexity formulas given in the previous section.

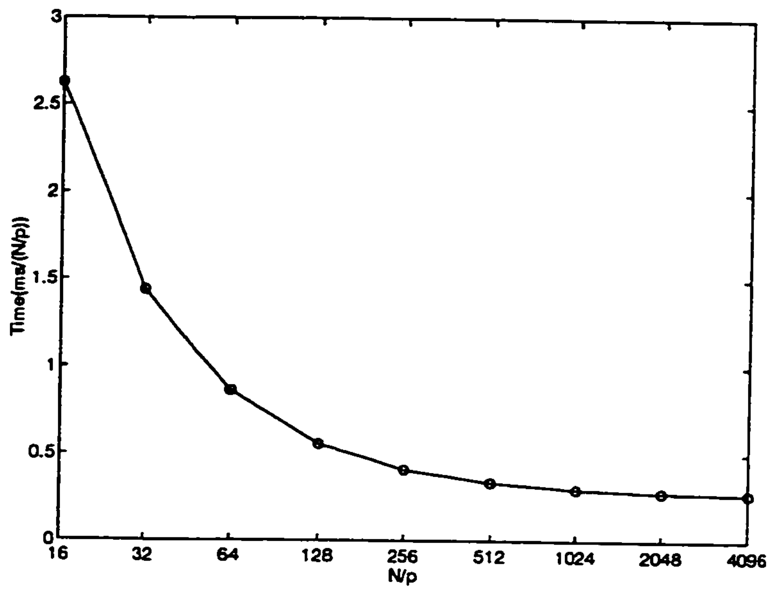


Figure 20: Performance of *A-select* on uniformly distributed data

Figure 20 shows the performance of *A-select* on uniformly distributed data and on different values of N/p. Figure 21 shows the performance of *A-select* on input distributions with varying entropies and $N/p = 1024$. We should notice here that the cost of the communications needed by *A-select* is very low (less than 4% of the total cost).

Figure 22 shows the relative performance for a uniformly distributed data of *A-select* with *A-ranksort*, *Radix sort*, and *B-flashsort*. The performance of our selection algorithm is much better (up to 25 times faster) than the performance of *Radix sort*, and up to 5.3 times faster than *B-flashsort*. It is reasonable to predict that *A-select* will perform very well on an MIMD computer knowing that only *b reduceAdd* operations are needed to select an element from the input keys.

Comparing the relative performance of *A-select* with *radix sort* implemented on MasPar MP-1 and the relative performance of the selection algorithm in [14] with *Radix sort* implemented on TMC CM-5, and by noticing that our selection algorithm needs only *b reduceAdd* operations to select an element from the input keys (compared to $O(2^r)$ *scan* operations needed by *Radix sort*), we can safely predict that *A-select* would have a better performance than the comparison-based selection algorithm in

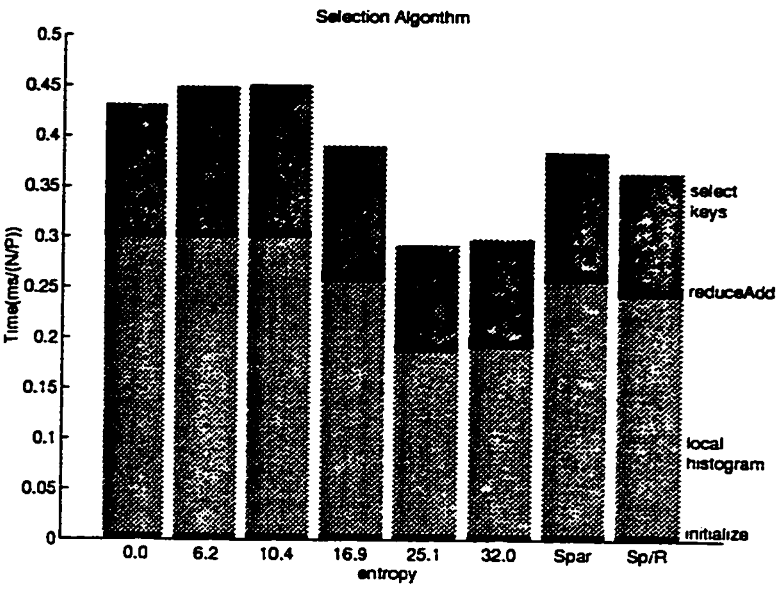


Figure 21: Performance of *A-select* for different entropies

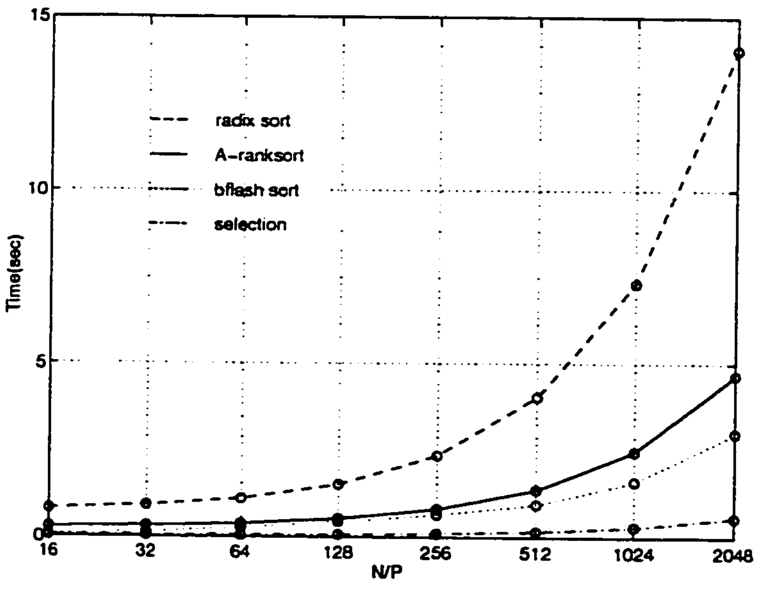


Figure 22: Comparison of *A-select* and sorting algorithms for uniformly distributed data

[14] on the TMC CM-5.

Chapter 5

Conclusions and Future Work

We have described a new parallel sorting algorithm, *A-ranksort*, which essentially consists of deterministically computing approximate ranks for the input keys, and using these to route to destinations that are guaranteed to be close to the final destination, and finally using a few steps of odd-even transposition sort to complete the sorting. Our algorithm is not a comparison-based algorithm; we use the bitwise representation of keys to aid in finding approximate ranks.

The advantages of our parallel sorting algorithm apart from its being the most efficient of known balanced algorithms are that it is conceptually simple, deterministic, and exhibits good performance for a wide variety of input distributions. It can be implemented on any parallel machine; it can work very efficiently in any machine where there is a cheap primitive for performing parallel prefix operations. *A-ranksort* can also be adapted to work for *any* sorting order, though our implementation is for row-major order and the performance is likely to degrade for some sorting orders. Our algorithm is based on an approximate ranking approach, which can be easily adapted to perform approximate selection.

The empirical results show that *A-ranksort* performance on data chosen uniformly at random improves slightly on the performance of *B-flashsort*, the best previously known algorithm, when the final output is required to be balanced. However, on input distributions of smaller entropies or sparse distributions, the performance of *A-ranksort* remains relatively stable, whereas the performance of *B-flashsort* degrades very badly. *A-ranksort* is the best choice for a variety of input distributions. Specifically, when the entropy of the data distribution is small, or when the data is chosen from a sparse set, our algorithm significantly outperforms *B-flashsort* as well as other known algorithms. Proving exact performance bounds for our sorting algorithm on different input distributions would be an interesting avenue for future research.

Our analysis suggests that the performance of our algorithm would improve on a machine like the CM-2 since it has a very fast scan operation, and the term in the running time to do with scans would be reduced for *A-ranksort* and *Radix sort*, while the other algorithms considered here will not get a corresponding performance benefit from this. Also, a machine with fewer but more powerful processors might allow a choice of r smaller than the one chosen here, and thus reduce the time for scans as well as for the final sorting phase. Verifying these ideas is beyond the scope of this thesis, but would be an interesting avenue of further research.

Our techniques for computing approximate ranks can also be used for other ranking problems, such as selection. In Chapter 4, we described a selection algorithm, *A-select*, that is based on the binary representation of the input keys. Our integer selection algorithm is simple and fast. For N b-bit keys and p processors, the local computation time complexity is $O(N/p)$ and it uses only b *reduceAdd* operations to find the sth element of the input data. An important characteristic of *A-Select* is that there is no data movement. We implemented our algorithm on MasPar MP-1. Our experimental results show that the relative performance of *A-Select* with *Radix sort* implemented on MasPar MP-1 is much better than the relative performance of the selection algorithm in [14] with *Radix sort* implemented on TMC CM-5. By noticing that our selection algorithm needs only b *reduceAdd* operations to select an element from the input keys (compared to $O(2^r)$ *scan* operations needed by *Radix sort*), we can safely predict that our algorithm would have a better performance than the comparison-based selection algorithm in [14] on the TMC CM-5.

It would be interesting to design an algorithm for a more general selection problem, where we find a set of keys in the input set of data (e.g find the keys with ranks $s_1, s_2, \ldots, s_i$). Future work may include developing and implementing algorithms for such problems based on our techniques for computing approximate ranks used in *A-ranksort* or based on a generalization of our selection algorithm.

Bibliography

[1] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[2] S. Albers and T. Hagerup. Improved Parallel Integer Sorting Without Concurrent Writing. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 463-472, 1992.

[3] A. Anderson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in Linear Time? In *ACM STOC'95*, Las Vegas, Nevada, USA, pp. 427-436, 1995.

[4] M. Ajtai, J. Komlos and E. Szemeredi. An $O(n \log n)$ Sorting Network. *in Proc. ACM Symp. Theory Comput.*, Apr. 1983, pp. 1-9.

[5] D. Angluin and L. Valiant. Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings. *Journal of Computer and System Science*, 18:155–193, 1979.

[6] K. Batcher. Sorting Networks and their Applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[7] K. E. Batcher. Design of the Massively Parallel Processor. *IEEE Trans. Comput.* C-29, 9 (Sept. 1980), 836-840.

[8] G. E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, vol. 38, no. 11, Nov. 1989.

[9] T. Blank. The Maspar MP-1 Architecture. In *Proceedings of IEEE Compcon spring.* IEEE, 1990.

[10] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS Parallel Benchmark Results 10-94. Report NAS-94-001, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, October 1994.

[11] D. W. Blevins, E. W. Davis, R. A. Heaton, and J. H. Reif. Blitzen: A Highly Integrated Massively Parallel Machine. *J. Parallel Distrib. Comput.* 8,2 (Feb. 1990), 150-160.

[12] P. Berthome, A. Ferreira, B. M. Maggs, S. Perennes, and C. G. Plaxton. Sorting-Based Selection Algorithms for Hypercubic Networks. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 89-95, Newport Beach, CA, April 1993. IEEE Computer Society Press.

[13] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7(4):448-461, 1973.

[14] D. A. Bader, and J. JaJa. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection. Technical Report 95. UMIACS. 1995.

[15] G. Blelloch, C. Leiserson, B. Maggs, G. Plaxton, S. Smith, and M. Zagha. A Comparison of Sorting Algorithms for Connection Machine CM-2. In *Symposium on Parallel Algorithms and Architecture*, pages 23–16. ACM, 1991.

[16] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *Annals of Mathematics and Statistics*, 23:493–507, 1952.

[17] P. Christy. Software to Support Massively Parallel Computing on the Maspar MP-1. In *Proceedings of the IEEE Compcon.* IEEE, 1990.

[18] B. Chlebus and M. Kukawka. A Guide to Sorting on the Mesh-Connected Array. Technical report, Instystut Informtyki, Warsaw, 1990.

[19] D. Culler, R. M. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *in Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 1-12, Jan. 1993

[20] T. H. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press and McGraw Hill, 1990.

[21] M.J. B. Duff. Review of the CLIP Image Processing System. *Proceedings of the National Computing Conference.* 1978, p. 1055-1060.

[22] S. Dasgupta. *Computer Architecture: A Modern Synthesis, Foundations.* John Wiley and Sons, Inc. , 1989.

[23] A. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Technical Report UCB//CSD-94-829, University of California-Berkeley, 1994.

[24] R. Diekmann, J. Gehring, R. Luling, B. Monien, M. Nubel, and R. Wanka. Sorting Large Data Sets on a Massively Parallel System. In *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 2–9, 1994.

[25] T. Dachraoui and L. Narayanan. Fast Deterministic Sorting on Large Parallel Machines. *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing* (SPDP'96), October 1996, pages 273–280.

[26] C. C. Elgot and A. Robinson. Random Access Stored Program Machines. *J. ACM*, vol. no. 4, pp. 365-399, 1964.

[27] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. on Computers C-21*, pp. 948-960, 1972.

[28] M. J. Flynn. Very High Speed Computing Systems. *Proc. IEEE* vol. 54, 1986, 1901-1909.

[29] R. W. Floyd, and R. L. Rivest. Expected Time Bounds for Selection. Communication of the ACM, 18(3):165-172, 1975.

[30] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. *in Proc. ACM Symp. Theory Comput.*, 1978, pp. 114-118.

[31] L. M. Goldschlager. A Universal Interconnection Pattern for Parallel Computers. *J. ACM*, vol. 29, no. 3, pp. 1073-1086, 1982.

[32] W. K. Giloi. Towards a Taxonomy of Computer Architecture Based on the Machine Date Type View. *Proc. of the 10th Annual International Symposium*

on Computer Architecture, IEEE Computer Society Press, New York, 1983, pp. 6-15

[33] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing a MIMD, Shared-Memory Parallel Machine. *IEEE Trans. Comput.*, vol. C-32, pp. 175-189, 1983.

[34] W. Handler. Standards, Classification and Taxonomy: Experiences with ECS. *in Blaauw and Handler* (1981), pp. 39-75.

[35] E. Hao, P. D. MacLenzie, and Q. F. Stout. Selection on the Reconfigurable Mesh. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, pages 38-45, McLean, VA, October 1992. IEEE Computer Society Press.

[36] C. A. R. Hoare. Algorithm 63 (partition) and Algorithm 65 (find). *Communications of the ACM*, 4(7):321-322, 1961.

[37] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.

[38] D. R. Helman, D. A. Bader, and J. JaJa. A Practical Sorting Algorithm with an Experimental Study. Technical Report 95-102, UMIACS, 1995.

[39] Y. Han, Y. Igarashi, and M. Truszczynski. Indexing Schemes and Lower Bounds for Sorting on a Mesh-Connected Computer. Technical Report, University of Kentucky, Lexington, 1988.

[40] W. Hightower, J. Prins, and J. Reif. Implementations of Randomized Sorting on Large Parallel Machines. In *Symposium on Parallel Algorithms and Architecture*, pages 158–167. ACM, 1992.

[41] D. J. Hunt. The ICL DAP and its Application to Image Processing. *in M. J. B. Duff, and S. Levialdi (Eds.). Language and Architectures for Image Processing.* Academic Press, London, 1981, pp. 275-282.

[42] J. F. JaJa. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992

[43] J. F. JaJa, and K. W. Ryu. The Block Distributed Memory Model. Technical Report CS-TR-3207, Computer Science Department, University of Maryland, College Park, January 1994.

[44] J. F. JaJa, and K. W. Ryu. The Block Distributed Memory Model for Shared Memory Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 752-756, Cancun, Mexico, April 1994. (Extended Abstract).

[45] D. Knuth. *The art of Computer programming, Vol. 3 (Sorting and Searching.* Addison-Wesley, 1973.

[46] M. Kunde. Lower Bounds for Sorting on Mesh-Connected Architectures. *Acta Informatica*, 24:121-130, 1987.

[47] M. Kunde. Routing and Sorting on Mesh-Connected Arrays. *In Aegean Workshop on Computing: VLSI algorithms and Architecture.* Vol. 319 of Lecture Notes in Computer Science, pages 423-433, Springer-Verlag, 1988

[48] M. Kunde. l-selection and Related Problems on Grids of Processors. *Journal of New Generation Computer Systems*, 2:129-143, 1989.

[49] M. Kunde. Concentrated Regular Data Streams on Grids: Sorting and Routing Near to the Bisection Bound. In *Symposium on the Foundations of Computer Science*, pages 141–150. IEEE, 1991.

[50] C. Kakalamanis and D. Krizanc. Optimal Sorting on Mesh-Connected Processor Arrays. In *Symposium on Parallel Algorithms and Architecture*, 1992.

[51] C. Kaklamanis, D. Krizanc, L. Narayanan, and A. Tsantilas. Randomized Sorting and Selection on Mesh-Connected Processor Arrays. In *Symposium on Parallel Algorithms and Architecture*, pages 17–28, 1991.

[52] D. Kirkpatrick and S. Reisch. Upper Bounds for Sorting Integers on Random Access Machines. *Theoretical Computer Science*, 28, pp. 263-276, 1984.

[53] R. V. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 17, pages 869-942. Elsevier, 1990.

[54] M. Kaufmann, J. Sibeyn, and T. Suel. Derandomizing Algorithms for Routing and Sorting on Meshes. In *Symposium on Discrete Algorithms*. ACM-SIAM, 1994.

[55] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes.* Morgan Kaufmann, 1992.

[56] Z. Li, P. H. Mills, J. H. Reif. Models and Resource Metrics for Parallel and Distributed Computation. *Proc. 28th Annual Hawaii International Conference on System Sciences (HICSS-28 Parallel Algorithms Software Technology Track)*, Wailea, Maui, Hawaii, January 3-6, IEEE Press, 1995.

[57] MasPar Computer Computation. *MasPar Parallel Application Language (MPL) Reference Manual.* 2nd beta edition, Doc #9302-0000-0690, Feb. 1990.

[58] S. Mullender. *Distributed Systems.* 2nd Edition, Addison-Wesley, 1993

[59] Y. Mansour, N. Nisan, and U. Vishkin. Trade-offs Between Communication Throughput and Parallel Time. *in Proc. of the 26th Annual ACM Symp. on Theory of Computing*, pp. 372-381, 1994.

[60] J. von Newmann. First Draft of a Report on EDVAC. Memorandum, reprinted in [R75], 355-364

[61] J. R. Nickolls. The Design of the MasPar MP-1: A Cost-Effective Massively Parallel Computer. In *Proceedings of the IEEE Compcon.* IEEE, 1990.

[62] L. Narayanan. Selectio, Sorting, and Routing on Mesh-Connected Processor Arrays. *Phd Thesis*, University of Rochester, Rochester, New York, 1992.

[63] J. F. Prins. Efficient Bitonic Sorting of Large Arrays on the MasPar MP-1. Proceedings of the 3rd Symposium on Frontiers of Massively Parallel Computation, pages 158-167, October, 1990.

[64] B. Randell. *Origins of Digital Computers.* Springer-Verlag, New York, 1975

[65] H. S. Stone. Parallel Processing with the Perfect Shuffle. *IEEE Trans. Comput.*, vol. C-20, no. 2, pp. 53-161, 1971.

[66] J. T. Schwartz. Ultracomputers. *ACM Trans. Programming Languages Syst.*, vol. 2, no. 4, pp. 484-521, Oct. 1980.

[67] H. J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* ed. 2. New York: McGraw Hill, 1990.

[68] H. J. Siegel et al. Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing. *Journal of Parallel and Distributed Computing* 16(3), pp. 199-211, 1992

[69] R. Sarnath and X. He. Efficient Parallel Algorithms for Selection and Searching on Sorted matrices. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 108-111, Beverly Hills, CA, March 1992. IEEE Computer Society Press.

[70] C. Schnorr and A. Shamir. An Optimal Sorting Algorithm for Mesh-Connected Computers. *In Symposium on the Theory of Computation*, pages 255-263, 1986.

[71] Y. Shiloach and U. Vishkin. Finding the Maximum, Merging and Sorting in a Parallel Computation Model. *Journal of Algorithms*, vol. 2, no. 1, pp. 88-102, 1981.

[72] C. Shannon and W. Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, 1949.

[73] K. Thearling. An Evaluation of Sorting as a Supercomputing Benchmark. Technical report.

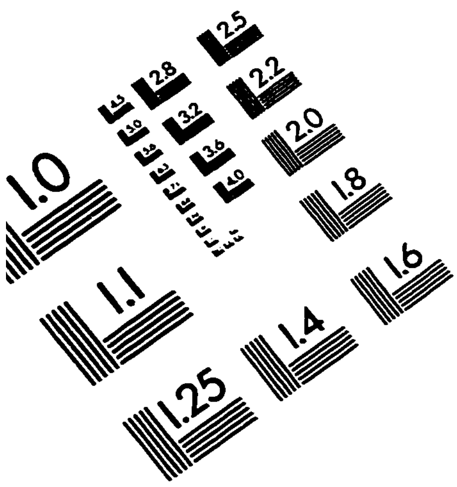
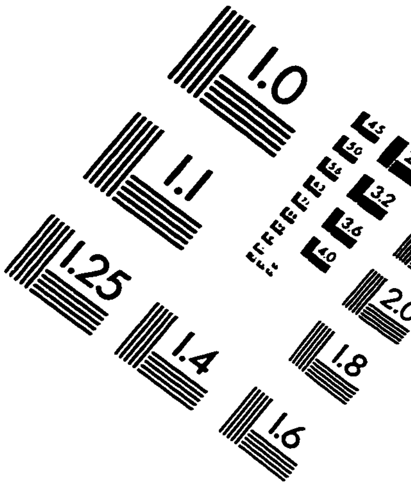
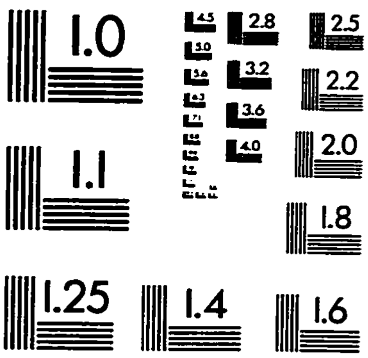
[74] C. Thompson and H. Kung. Sorting on a Mesh Connected Parallel Computer. *Communications of the ACM*, 20:263–270, 1977.

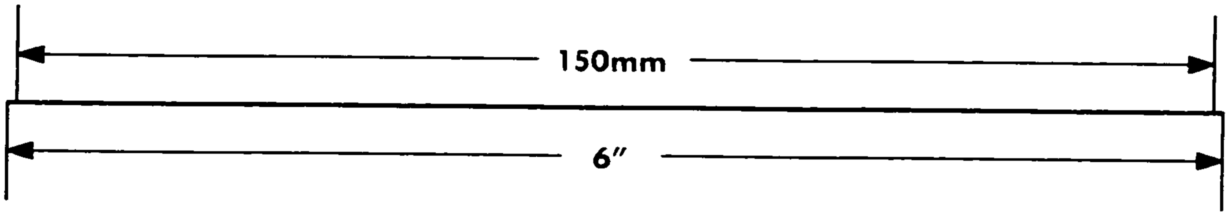
[75] K. Thearling and S. Smith. An Improved Supercomputing Sorting Benchmark. In *Supercomputing.* ACM, 1992.

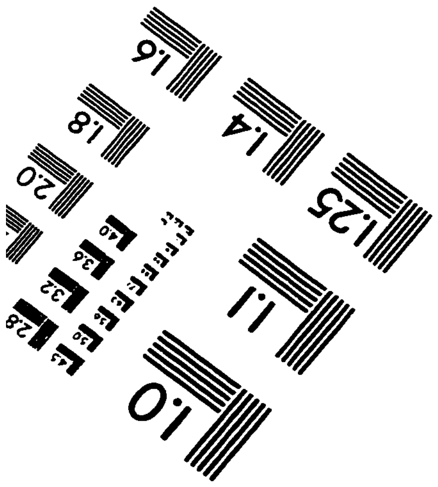
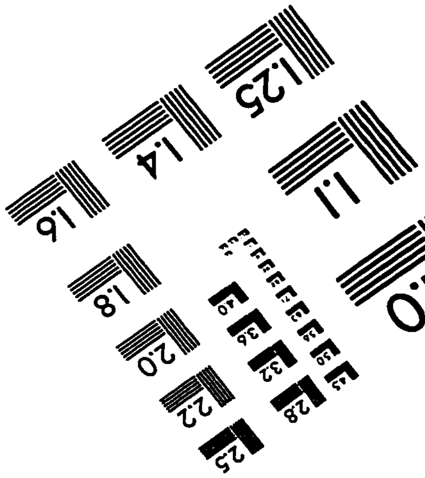
[76] L. Valiant. A Bridging Model of Parallel Computation. *Communications of the ACM*, 33(8), pp. 103-111, Aug. 1990.

[77] A. Y. Zomaya. *Parallel and Distributed Computing Handbook* Albert Y. Zomaya. Editor, McGraw-Hill inc., 1996.

IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989