

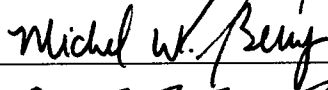
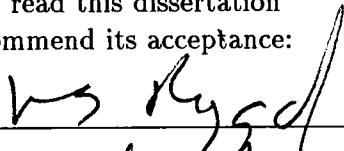
To the Graduate Council:

I am submitting herewith a dissertation written by Clay Patrick Breshears entitled "Adapting Non-Numeric, Parallel Algorithms for Diverse Models of Computation." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.



Dr. Michael A. Langston, Major Professor

We have read this dissertation
and recommend its acceptance:



Accepted for the Council:



Associate Vice Chancellor
and Dean of the Graduate School

Adapting Non-Numeric, Parallel Algorithms for Diverse Models of Computation

A Dissertation
Presented for the
Doctor of Philosophy Degree
The University of Tennessee, Knoxville

Clay Patrick Breshears
August 1996

Copyright 1996 by Clay P. Breshears
All rights reserved

DEDICATION

This thesis is dedicated to my wife, Lorna, for her patience and continued support even after it took much longer to finish than I'd originally promised.

ACKNOWLEDGMENTS

I am indebted to my advisor, Dr. Michael A. Langston. I thank him for the time and effort he spent working with me, the opportunities for research and professional development he provided and the patience he displayed when trying to make me realize some misconception or error in logic on my part.

I am also grateful to the rest of my committee, Dr. Michael Berry, Dr. Mark Jones and Dr. W. Stuart Riggsby. Their questions and comments on the manuscript helped to improved the quality of this dissertation.

I would like to thank Steve Moulton for his friendship and technical expertise, Judi Theg Talley for sharing her $\text{\LaTeX}$ expertise, Dr. Christian Halloy and the entire staff, past and present, of the Joint Institute for Computational Science for their support, knowledge and advice on the use of such a large number of parallel machines, and Geoffrey von Limbach for showing me that finding simple solutions to complex problems may only be a matter of perspective.

For everything they gave to me when growing up and their encouragement and support that got me to this point in my life, I wish to thank my parents, Howard and Jane Breshears. I especially acknowledge my father, who saw me start the process but is no longer with us to brag about having a doctor in the family.

Finally, I thank my wife, Lorna. Without her gentle pestering (often as a tag team partner with my advisor), I might never have finished this dissertation.

ABSTRACT

The simplicity of the PRAM model avoids communication latency, memory limitations and other difficult issues. Efforts have been made to develop more realistic models, especially with respect to communication, yet the restriction of finite memory space on computer architectures remains virtually unaddressed. Models cannot encompass all aspects of the hardware and the programming of parallel machines without losing much of their simplicity and usefulness. However, when implementing non-numeric algorithms (e.g., merging, sorting, graph algorithms), efficient use of available memory can be a major design concern.

The transport of code from one machine model to a different model presents the programmer with a number of challenges. Parallel numeric computations are generally very regular with respect to memory access, processor cooperation and ordering of computations. The efficient implementation of such algorithms is intimately tied to the executing architecture and, consequently, may require a large amount of redesign when moved to a different machine. Non-numeric computations are typically driven by the values of the data itself. Prediction of memory access patterns and processor cooperation *a priori* is nearly impossible. The unpredictable nature of non-numeric computations does not often allow one to take advantage of the architectural constraints. Without such benefits, the design and porting of non-numeric algorithms between machines can be simpler (though less efficient) than for numeric algorithms. Even so, many practical considerations and obstacles cannot be anticipated from mod-

els.

We investigate the process of transporting non-numeric algorithms between diverse machine models. For this study we have chosen a time-space optimal parallel merge as our prototypical algorithm. This merging strategy possesses many algorithmic properties shared by a wide range of other parallel non-numeric algorithms. The merge algorithm has the additional property of requiring only a constant number of auxiliary memory cells beyond that needed to hold the data for merging, effectively addressing the finite bound of memory on practical machines. As a result of the variety of architectures on which the merge algorithm is implemented, we also examine the use of the merge code as a non-numeric parallel benchmark. Based on these experiences, we develop improvements to reduce the amount of communication in the merge algorithm. In all cases, actual implementations and executions of programs are described.

Contents

1	Introduction	1
1.1	Theoretical versus Practical Computation	1
1.2	Non-Numeric and Numeric Computation	3
1.3	A Prototypical Non-Numeric Problem	4
1.4	Overview	6
2	Review of Time-Space Optimal Merging and Machine Details	8
2.1	Introduction	8
2.1.1	Algorithmic Conventions	8
2.2	Parallel In-Place Merging	9
2.3	Serial Time-Space Optimal Merging	13
2.4	Machine Details	14
2.4.1	Thinking Machines CM-2	14
2.4.2	Thinking Machines CM-5	14
2.4.3	Sequent Symmetry S81	15
2.4.4	MasPar MP-2	15
2.4.5	Intel iPSC/860	16
2.4.6	Intel Paragon	16
2.4.7	IBM SP2	16
3	MIMD to SIMD Conversion	18
3.1	Introduction	18
3.2	An MIMD Implementation	19
3.2.1	Implementation Details	19
3.2.2	Computational Experience on the S81	22
3.3	Extension to the SIMD Model	24
3.3.1	C* Code Conversion	24
3.3.2	Computational Experience on the CM-2	29
3.3.3	MPL Code Conversion	30
3.3.4	Computational Experiences on the MP-2	32
3.4	Interpretation of Results	34
3.4.1	Initial Remarks	34

3.4.2	Data Locality and Cache Effects	35
3.4.3	Single Instruction Limitations	37
3.4.4	Task Synchronizations	39
3.5	Some Lessons Learned	40
3.5.1	Architectural Performance	41
4	Shared Memory to Distributed Memory Conversion	43
4.1	Introduction	43
4.2	A Simple Technique	45
4.3	Conversion Details	49
4.3.1	CMMD Advantages	49
4.3.2	Examples	51
4.3.3	Displacement Table Computation	54
4.4	Computational Experience on the CM-5	56
4.4.1	Message Size Variations	58
4.4.2	Timing by Step	59
4.5	Interpretation of Results	60
4.5.1	Irregular Communication	60
4.5.2	Consumer/Producer Delays	61
4.5.3	Block Size Variations	62
4.6	Some Lessons Learned	64
4.6.1	Other Machines	66
5	Parallel Benchmarks and Comparison-Based Computing	67
5.1	Introduction	67
5.2	Previous Efforts	68
5.2.1	Sequential Benchmarks	68
5.2.2	Parallel Benchmarks	70
5.2.3	Database Benchmarking	72
5.2.4	In Summary	73
5.3	Defining a Suitable Benchmark Algorithm	74
5.3.1	Non-Numeric Operations	75
5.3.2	The NAS Parallel Sort	76
5.3.3	Radix Sort	77
5.3.4	Valiant's Parallel Merge	79
5.3.5	In-Place Merging	80
5.4	Implementation and Timing Results	81
5.4.1	Methodology	82
5.4.2	Machine Comparisons	82
5.4.3	Memory Capacity	86
5.4.4	Benchmark Comparisons	86
5.5	Discussion	88

6	Algorithmic Improvements	89
6.1	The Problem	89
6.2	SIMD Algorithmic Improvements	90
6.2.1	First Method	91
6.2.2	Second Method	92
6.2.3	Analysis	93
6.2.4	Timing Results	94
6.3	Displacement Table Computation	96
6.3.1	Timing Results	99
6.4	Combining Block Sorting and Record Movement	100
6.4.1	$O(n)$ Extra Memory Version	100
6.4.2	$O(1)$ Extra Memory Version	104
6.4.3	Implementation Details	111
6.4.4	Timing Results	112
6.5	Summary	114
7	Conclusions and Future Work	116
7.1	Contributions	116
7.2	Open Questions	118
7.3	Code Availability	119
	BIBLIOGRAPHY	120
	VITA	126

List of Figures

1.1	OVERVIEW OF RESULTS	6
3.1	S81 TIMING RESULTS (16 PROCESSORS)	24
3.2	CM-2 TIMING RESULTS (16384 PROCESSORS)	31
3.3	MP-2 TIMING RESULTS (4096 PROCESSORS)	33
4.1	CM-5 EXECUTION TIME	57
5.1	DETAILED PERFORMANCE OF EACH MACHINE	85
5.2	RANGE OF EXECUTION TIMES FOR SELECTED MACHINES	85
6.1	FIRST METHOD ILLUSTRATION	91
6.2	SECOND METHOD ILLUSTRATION	93
6.3	SIMD LOCAL MERGE TIME FOR IMPROVED ALGORITHM (ON MP-2)	96
6.4	ONE MOVE TIME VERSUS TWO MOVE TIME (ON CM-5)	114

List of Tables

3.1	TOTAL S81 EXECUTION TIME	23
3.2	SAMPLE S81 STEP TIMES	23
3.3	TOTAL CM-2 EXECUTION TIME	30
3.4	SAMPLE CM-2 STEP TIMES	30
3.5	TOTAL MP-2 EXECUTION TIME	32
3.6	SAMPLE MP-2 STEP TIMES	33
4.1	TOTAL CM-5 EXECUTION TIME (32 PROCESSORS)	56
4.2	COMPARISON OF CM-5 AND S81 EXECUTION TIMES	57
4.3	CM-5 RUN TIMES WITH DIFFERING MAXIMUM MESSAGE SIZES . .	58
4.4	SAMPLE CM-5 STEP TIMES	60
4.5	COMMUNICATION TIMES BY STEP	63
4.6	NON-COMMUNICATION TIMES BY STEP	63
5.1	MACHINES USED IN STUDY	81
5.2	AVERAGE PERFORMANCE OF EACH MACHINE	84
5.3	MEMORY CAPACITY MEASURE AND PERFORMANCE	86
5.4	COMPARISON OF LINPACK AND G-L MERGE BENCHMARKS . . .	87
5.5	COMPARISON OF NAS INTEGER SORT AND G-L MERGE BENCHMARKS	87
6.1	CM-5 SIMD IMPROVED LOCAL MERGE TIMING	95
6.2	MP-2 SIMD IMPROVED LOCAL MERGE TIMING	95
6.3	STEP 3 TIMING COMPARISON (ON CM-5)	99
6.4	SINGLE DATA MOVE TIMING COMPARISON (ON CM-5; BLOCKS- IZE = 1)	113
6.5	SINGLE DATA MOVE TIMING COMPARISON (ON CM-5; BLOCKS- IZE ≤ 100)	113

Chapter 1

Introduction

1.1 Theoretical versus Practical Computation

The Parallel Random Access Machine (PRAM) model is the *de facto* standard theoretical model for non-numeric parallel algorithms. This model and its properties have been studied extensively in the literature [21, 37, 44]. The PRAM model uses a potentially unbounded number of processors executing synchronously with access to a shared global memory. Communication, memory space and other difficult issues are ignored in this simple model. Because the number of processors is some function of the input size, say $f(n)$, execution time scales trivially as the data size increases.

While the PRAM model may be useful for the design and analysis of parallel algorithms, direct implementations of PRAM-style algorithms on real parallel machines require some extra thought and programming. Unlike the PRAM model, real parallel machines have a finite number of processors. This difference can be easily overcome by using each physical

processor to emulate a portion of the processors prescribed by the algorithm. For instance, given a PRAM model algorithm that requires $f(n)$ processors and a parallel machine with p processors, each step of the actual computation would require each physical processor to execute the same statements $f(n)/p$ times, once for each simulated or “virtual” processor. Such an emulation also requires that each virtual processor’s registers and local variables be held in the memory of the physical machine. Unlike the PRAM model, real parallel machines have a finite amount of memory. Thus, the trivial scaling achievable on the PRAM model soon runs into a real hardware limitation.

Karp and Ramachandran [37] call for the study of bounded processor models. This modification to the PRAM model easily overcomes the scaling problem stated above by forcing algorithm designers to work within more practical machine constraints. As a benefit of this approach, algorithms designed on processor-bounded models scale well when implemented on real parallel machines.

Many recently proposed models of parallel computation seek to bridge the gap between theoretical algorithms and their implementations by modeling actual machine characteristics better than the PRAM model does. While all of these new models utilize a fixed number of processors unrelated to the size of the input, their primary focus has been asynchronous processor execution [13], network and memory latency [3, 15, 59] or hierarchical memory [61]. The amount of memory within these theoretical models remains unbounded.

A PRAM algorithm is defined to be optimal in terms of run time alone; the efficient use of memory space is not addressed. Yet, as evidenced by the virtual processor PRAM

emulation scheme on real machines, space usage is an important consideration. Development of a space-bounded model is a solution. It is our thought, though, that the arbitrary selection of a finite amount of memory as part of a model's specification would needlessly limit the model. It is more general to build such considerations directly into the algorithm.

1.2 Non-Numeric and Numeric Computation

Matrix and related linear algebra computations continue to be a major impetus behind numeric parallel algorithm development. As the availability of parallel computers expands into more mainstream venues, increasing emphasis is being placed on research and development of non-numeric parallel algorithms; e.g., merging, sorting, graph algorithms.

The differences between non-numeric and numeric computations are many. Non-numeric algorithms tend to be composed of *compare-move* operations, while numeric calculations generally perform *access-compute-store* operations. That is, non-numeric algorithms tend to be based around the operation of comparing two data items and moving data from one location to another based upon the results of that comparison. Numerical algorithms, on the other hand, are composed of arithmetic operations on two values followed by the storage of the result in a location determined by the algorithm. Another way to look at this difference is that non-numeric computations are dynamically driven by the values of data items; whereas numeric computations are statically predetermined by the algorithm and structure of the data. Some algorithms (e.g., those dealing with unstructured meshes and sparse matrices), though the calculations are predominantly numeric, could be argued to fit under the non-numeric

classification since the execution of instructions and movement of data are often determined by the values and positions of data.

In terms of parallel algorithms, the cooperation of tasks within non-numeric codes is very unpredictable at the outset. This unpredictability is also found in the communication patterns between processors and in local memory access. Numeric codes tend to be more predictable in all of the above respects. Parallel processing is still a relatively new endeavor. One goal of parallel algorithm research is the emergence of a single model for the design and analysis of algorithms that can be implemented across many parallel machines. Non-numeric algorithms are well suited for examining the reality of models and testing the portability of codes between machines. Such program transformations often give us an insight into the width of the gap between the theory and the practice of parallel programming.

1.3 A Prototypical Non-Numeric Problem

We will study the adaptation and transporting of non-numeric parallel algorithms from one computing platform to another: from MIMD (Multiple Instruction, Multiple Data) to SIMD (Single Instruction, Multiple Data) models and between shared and distributed memory machines. Rather than code, implement and test dozens of different non-numeric algorithms, a representative algorithm that embodies the computations and algorithmic properties of a wide range of non-numeric algorithms was selected. We chose the in-place parallel merge of Guan and Langston [28] as our prototypical algorithm. This algorithm, henceforth termed the G-L merge, requires a non-trivial implementation and exhibits a variety of data movement

patterns—both locally and globally—and processor interactions that are the hallmarks of typical non-numeric algorithms.

It may be argued that the problem of sorting is more typical of non-numeric computing. Sorting has been extensively studied and there exist many different parallel algorithms designed to solve this problem. This diversity makes it hard to choose one particular sorting algorithm as a representative of non-numeric computations. Parallel sorting algorithms often directly rely on the underlying processor interconnection network [17, 48, 51]. Implementing such a sorting algorithm onto a machine with different network connections would unduly inhibit the efficient execution of the algorithm. In many ways, therefore, merging is a more general problem than sorting, and is more easily adapted to a wide variety of parallel machines. Moreover, one can always use a merge algorithm to construct an efficient sort with a straightforward sort-by-merging approach.

The PRAM algorithm for the G-L merge makes use of only a constant amount of auxiliary memory beyond that needed for the data. Thus, the bounded memory restriction found on real machines is addressed, and good scaling properties are achieved in practice. A fortuitous side effect of using such an in-place algorithm is the ability to load primary memory to capacity on any realized parallel hardware. This gives us the opportunity to study the performance of parallel machines at full data capacity.

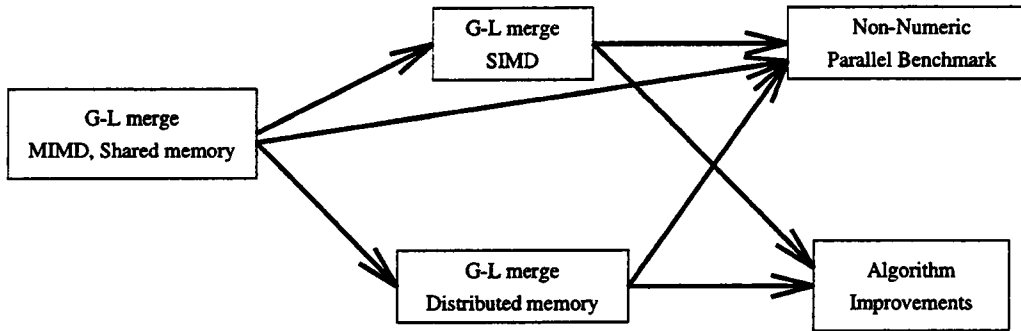


Figure 1.1: OVERVIEW OF RESULTS

1.4 Overview

Figure 1.1 gives a pictorial overview of our results. After a review of time-space optimal merging in Chapter 2, we examine the conversion of non-numeric parallel codes from one machine to another. The G-L merge, implemented initially on a shared memory MIMD architecture, is ported first to an SIMD model (Chapter 3) and then to a distributed memory model (Chapter 4). For each programming and execution model considered, we uncover several features and pitfalls of real program implementation unanticipated from the design of the original algorithm on a Exclusive-Read, Exclusive-Write (EREW) PRAM model.

With the same algorithm adapted to many different architectures, we demonstrate the feasibility of using the G-L merge code as a non-numeric parallel benchmark in Chapter 5. This includes a survey of current non-numeric benchmark codes and compares the results of the G-L merge against other currently used parallel benchmarking suites in order to gauge how well these suites can predict the execution performance of non-numeric codes. From our experiences with each of the different parallel architectures, several techniques for reducing

the amount of communication and computation are developed and discussed in Chapter 6.

In Chapter 7, we present a summary of this effort.

Chapter 2

Review of Time-Space Optimal Merging and Machine Details

2.1 Introduction

In this chapter we detail the highlights of both the parallel and serial in-place merge algorithms. Pertinent details of the parallel machines used in our studies are also included.

2.1.1 Algorithmic Conventions

The G-L merge is designed for the EREW PRAM model. Unlike other PRAM merges, it anticipates a fixed number of processors, p . We assume these processors are numbered consecutively from 0 to $p - 1$. The notation P_i is used to refer to the processor with index number i . The n data records to be merged are assumed to be arranged linearly and indexed from 0 to $n - 1$. (For implementations, we use an array named A to hold the records. On distributed memory machines, the data will be divided into blocks. Each processor stores its portion of the data in a local array A indexed from 0 (zero) to hi .)

During the course of the merge, the data is divided into contiguous blocks. With regard

to the relative positions of processors, blocks and records, those items with larger index values are considered to be located to the *right* of those with lower values. Similarly, the notion of processors, blocks and records preceding or following another is also based on index values. The *head* of a block is the leftmost (lowest indexed) record within the block. The *tail* is the rightmost (highest indexed) record within the block.

2.2 Parallel In-Place Merging

The time-space optimal parallel merge developed by Guan and Langston is robust, ensuring time-space optimality even on the weak EREW PRAM model. In what follows, we give a high-level description of this method, and assume a file with a total of n records is input containing two sorted subfiles to be merged. For simplicity, we assume each of the two sublists is evenly divisible by p , the number of processors. We refer the reader to [28] for a complete description of the algorithm and attendant special cases.

The merge algorithm is comprised of five steps. The first four reduce the problem to one of local merges at each processor (the last step). The data to be merged is viewed as p blocks, each block of size n/p , and each block managed by a distinct processor.

The first step is to sort the blocks by their tail key. This is accomplished by extracting a copy of each tail key and its corresponding processor (and, hence, the block number) index, and sorting these with any PRAM sort or merge algorithm. A bitonic merge [7] is easily implemented for the tail sort.

Upon completion of the tail sort, each processor will hold the index of that block whose

records will replace its own block. Before the actual movement of records begins, processors send their own processor index to that processor whose block index they hold from the tail key sort. Blocks of records are moved to the appropriate memory locations using these processor indices. The blocks of data are now said to be *tail-sorted*.

The second step divides the data into pairs of series. Boundaries of the series pairs are located at *breaker* records. Such breakers are the first record in block $i + 1$ whose key is no smaller than that of the tail of block i . Those records that precede a breaker within a block are designated as the second series records. The first series records are those immediately following a breaker (or the leftmost boundary of the data file) and preceding the block containing the subsequent breaker. While each set of second series records will be contained within a single block, first series records may span zero or several full blocks plus a portion of a block containing a breaker record; i.e., processors that contain breakers will have both a second series and a first series in their block of data.

When a processor determines that it has a breaker in its block, the index of the breaker is broadcast to all other processors. Thus, each processor can determine the boundaries of the series pairs around it and its own position within a series pair. Those processors holding only a first series block of records are known as *first series processors*; those holding a breaker are *second series processors*. Second series processors hold not only a second series block of records, but also a first series block located to the right of the breaker. All first series blocks are said to be *associated* with the leftmost second series block (processor) to the right within a series pair and vice versa.

The third step computes a *displacement table* entry, E_i , for each block. This value is the number of first series records from a block that would need to be exchanged with second series records in order to place records into the block in which their final merged position is located. Each processor containing first series records calculates its own displacement table entry as if there were no other first series blocks sharing the associated second series. To compute this value, processors holding first series records begin by performing a *phased merge*. The phased merge procedure repeatedly merges first series tail keys with selected keys from the associated second series block. Each subsequent merge selects new second series keys by subdividing the sub-block—delimited by two consecutive second series keys—by the number of first series keys that were merged between them. The subdividing and merging phases continue until each first series processor, P_i , has determined the position, l_i , within the second series its tail key would be located if merged into the second series block.

Processors now set up two index pointers: one at the tail of their first series block and the second, $p2_i$, at the l_i position of the associated second series. That pointer which indexes the larger key value is decremented until a total of l_i decrements have been performed. The offset of $p2_i$ into the second series becomes the displacement table entry E_i .

The fourth step distributes the records of the second series among the blocks of the associated first series as well as shifting displaced records between first series blocks. Before records are moved, the E_i rightmost first series records (those to be displaced) are moved to the leftmost positions of the first series block. This is done via sub-block rotations within first series blocks. The first series records to be displaced within a block holding a breaker

will be immediately to the right of the breaker position. The leftmost E_i first series records are then systematically moved from the block of P_i to the corresponding leftmost positions of the block of P_{i+1} . The exception to this is when P_i controls a second series block.

Given P_j holds the leftmost first series block within a series pair and P_k is the associated second series processor, as the first series records are being moved, the E_j second series records are moved into the vacated first series record positions within block j from second series block held by P_k . The next $E_{j+1} - E_j$ second series records are moved to block $j + 1$. Second series records continue to be moved in this way until the final $E_{k-1} - E_{k-2}$ second series records have been moved to first series block $k - 1$.

The fifth and final step is a time-space optimal merge on the two sorted sub-lists formed by the records moved into the block during the previous step and those records that remained in the block. If the length of the shorter list is greater than a threshold value, the sequential merge of Huang and Langston, described below, is used. Otherwise a simple block merge rotates unmerged portions of the shorter list with records from the longer list until the two lists are sorted.

It should be noted that processors with breaker records may need to perform separate merges for each sub-block of data located on either side of the breaker position. If a portion of the first series records—located to the right of the breaker—were displaced during the record movement (fourth) step, a merge of the two sub-lists to the left of the breaker and a merge of the two sub-lists to the right of the breaker will need to be performed.

2.3 Serial Time-Space Optimal Merging

This section gives a superficial overview of the algorithm for sequential linear time, in-place merging developed by Huang and Langston [33].

Given two sorted lists of n total records, each list is divided up into blocks of size $\lfloor \sqrt{n} \rfloor$. The $\lfloor \sqrt{n} \rfloor$ largest keyed records are collected into an internal buffer which is positioned ahead of the other data blocks. These remaining blocks are rearranged in sorted order by their tail key. The leftmost block whose head key is smaller than the preceding block's tail key is located. This block is designated as the *S2* (second series) block and all records preceding this block are *S1* (first series).

Records are merged from the *S1* and *S2* sublists into the positions occupied by the buffer records. This causes the buffer to "roll" through the unmerged records leaving a sorted list in its wake. The merge of *S1* and *S2* records continues until all *S1* records have been merged. The leftmost block within the remaining unmerged records whose head key is smaller than the preceding block's tail key is located. As before, this block becomes the (new) *S2* block while all preceding unmerged records, including the remaining records from the previous *S2* block, are now considered to be *S1*. These two series of records are merged as before.

The merge algorithm proceeds in this fashion until all records (except those in the buffer) have been merged. The buffer, containing the $\lfloor \sqrt{n} \rfloor$ largest records from the original lists, will be located in the rightmost positions of the data and is sorted by an efficient, in-place sort such as heapsort [14].

We have omitted many of the details and special cases that must be addressed when im-

plementing the above algorithm. The reader is referred to [33] for a more complete description of the algorithm and implementation details.

2.4 Machine Details

We now describe the hardware we have used. All our codes are written in the resident version of the C language augmented with any necessary communication libraries.

2.4.1 Thinking Machines CM-2

The CM-2 is an SIMD, data parallel computer containing as many as 65536 bit-serial processors. Groups of 16 processors are contained within a single chip as a processing node that communicates with other nodes in the system through a router network with a hypercube configuration. A standard serial computer acts as a front end processor executing the serial portions of the user's code and controlling the execution of the parallel code on the CM-2.

Operating System: SunOS Release 4.1.3.U1; CM System Software Release 6.1.2
Compiler: C* Version 6.0.3 (196) for sun4
Processors: proprietary, bit-serial
Memory: 64 Kbits per processor

2.4.2 Thinking Machines CM-5

The CM-5 is able to execute in either SIMD or MIMD mode. A data and separate control network, based on a fat tree topology [42], connect all processing nodes. The data network is used for data movement and interprocessor communication while the control network is involved in those operations in which all processors participate, such as synchronization and broadcasting. Up to four optional vector processing units may be attached to each processing

node.

SIMD Details:

Operating System: SunOS Release 4.1.2; CMOST Version 7.2
Compiler: C* Driver Version: 7.1 Final Rev: f2000
Processors: Sun SPARC1 (plus 4 vector units per node)
Memory: 32 Mbytes per node

MIMD details:

Operating System: SunOS Release 4.1.2; CMOST Version 7.2
Compiler: Standard SunOS C compiler
Processors: Sun SPARC1
Memory: 32 Mbytes per node

2.4.3 Sequent Symmetry S81

The Sequent Symmetry series of computers are tightly coupled MIMD multiprocessors with a common shared memory. The CPUs, memory modules and I/O controllers are all attached to a single high-speed bus. A maximum of 30 processors may be incorporated in a Model S81 system.

Operating System: DYNIX(R) V3.0
Compiler: Symmetry C, version 1.2
Processors: 80386 microprocessor
Memory: 32 MBytes

2.4.4 MasPar MP-2

The MP-2 is an SIMD machine with thousands of RISC processors connected via a 2-D mesh with wraparound. Instructions are issued to the processor array from the Array Control Unit. A front end computer is used to compile codes for the MP-2 and interface the machine with outside file servers and networks. A three stage global router is used for general, point-to-point communication patterns between processors.

Operating System: ULTRIX V4.3 (Rev. MP-3.22)
Compiler: MPL, Version 3.2.14
Processors: proprietary RISC
Memory: 64 Kbytes per processor

2.4.5 Intel iPSC/860

The iPSC/860 is a high performance MIMD parallel computer system using Intel i860 microprocessors connected in a hypercube configuration. A front end host is used to interface the processors with outside file servers and networks.

Operating System: iPSC/860 UNIX Sys V 3.2, NX 3.3.2
Compiler: icc/NX Sun4 Rel 4.0
Processors: Intel i860
Memory: 8 Mbytes per node

2.4.6 Intel Paragon

The Paragon supercomputer is based on Intel's teraFLOPS architecture. A 2-D rectangular mesh-connected network of processors distinguishes the Intel Paragon from the previous Intel supercomputer model's hypercube connection network. Each node comes equipped with a separate message processor to free the application processor from dealing with the handling of message traffic.

Operating System: Paragon OSF/1 Release 1.0.4 Server 1.3 R1.3
Compiler: icc/Paragon Version R5.0.1
Processors: Intel i860 XP
Memory: 16 Mbytes per node

2.4.7 IBM SP2

The SP2 is a loosely coupled collection of RISC 6000 superscalar processors. Each processor can be used individually or within an SPMD (Single Program, Multiple Data) or MIMD

parallel computation. An optional high performance switch may be installed to facilitate high-speed communication between processors. Nodes on the SP2 are designated to be either "thin" or "wide" dependent upon the amount of local memory and the number of microchannel slots available to the processor.

Operating System: AIX 6000
Compiler: AIX C Compiler with MPL Extensions
Processors: RS/6000 (POWER2 architecture)
Memory: 128 Mbytes per node

Chapter 3

MIMD to SIMD Conversion

3.1 Introduction

In this chapter we study design and implementation issues for non-numeric parallel algorithms on real MIMD and SIMD machines. To our knowledge there has been no other published case studies describing the explicit conversion of shared memory parallel codes to the data parallel paradigm. (We shall restrict our discussions to shared memory MIMD machines in this chapter and examine distributed memory machines in the sequel.)

Our representative MIMD machine is the Sequent Symmetry S81. The SIMD machines we have utilized are the Thinking Machines CM-2 and the MasPar MP-2. The S81 employs a handful of fairly powerful processors. The CM-2 uses tens of thousands of relatively simple processors; the MP-2 system contains thousands of proprietary RISC processors. The S81 connects its processors to a shared memory through a common bus. The CM-2 connects its processors in a hybrid binary hypercube fashion with memory distributed among the individual processors; the MP-2 networks processors in a 2-dimensional toroidal mesh with general point-to-point communication facilitated through a three-stage routing network.

We develop techniques that have been helpful for us and that may be useful in future conversions of this nature. In addition to our discussion of implementation issues, we present sample timing results from all three machines. We emphasize that our goal is not to fine-tune our codes to make them competitive with microcode implementations available elsewhere, but rather to discover stumbling blocks inherent in the MIMD-to-SIMD program transformation process. Thus these timing figures, slow though they may be, are useful in identifying the relative strengths and weaknesses of these two distinct models.

3.2 An MIMD Implementation

3.2.1 Implementation Details

Having been designed with the shared memory MIMD model in mind, the implementation of the G-L merge onto the S81 was not difficult. The code was written in the C language augmented with Sequent Parallel Programming Library routines needed to perform data sharing and synchronization.

Two categories of variables can be used: *shared* or *private*. The data to be merged and the auxiliary variables whose values are to be used by more than a single processor are declared to be shared. All others are private and treated as if they were local to each processor. Each processor knows its own identification number in relation to all other processors; consequently, the indices for the boundaries on the blocks of the files that are under any one processor's control are easily calculated.

PRAM models typically assume a lockstep execution of instructions ([4, 37, 39]). In

real MIMD machines, no such lockstep is usually enforced. Each processor is given its assignment and allowed to run the instructions asynchronously. Should a sharing of data or other cooperation between tasks on separate processors be necessary, the program must provide some method of synchronization between the cooperating processors. For the parallel merge implemented on the Sequent Symmetry there is a large amount of cooperation required among processors within the first four steps of the algorithm. Many portions of the code require processors to use the values of pointers and counters held by a neighboring processor. Nearly every one of these types of access must be preceded and/or followed by a global barrier synchronization to ensure that processors do not violate EREW restrictions. The choice of barrier synchronization over other methods is based on availability and programming simplicity. Subsequent coding of portions of the merge algorithm using locks in place of barriers resulted in much slower execution times.

The barrier synchronization `m_sync` is used to synchronize all processors to a single point in the code. After the processors have executed the barrier synchronization command, all processors are allowed to proceed with their next instruction. The coordination of the synchronizations can be a chore in itself for the programmer. Since each call to `m_sync` is identical to every other call, processes can actually be synchronized at different points in a program. For example, after the blocks are sorted by their tails and the breakers are found among the blocks, any blocks that are situated after the final breaker are already in sorted order and the processors that are handling those blocks need not participate in any following execution. Once these processors are identified, it is an easy matter to ensure

that they do not execute any further code that may move the records they are holding and consequently unsort them. However, because the remaining active processors must cooperate among themselves, there still remains a number of synchronization points that must be met and passed. If reciprocal `m_sync` instructions are not programmed in for those processors that have finished processing early, the synchronization of the whole program will be compromised and the system can easily become deadlocked. Adding more processors to the computation only magnifies the difficulty of tracking down and debugging this type of error.

Since calls to `m_sync` involve a function call and its attendant overhead, it is tempting to omit a synchronization point to reduce execution time, especially when you have code of the form

```
m_sync();
num = blanks[myid-1];
m_sync();
if (temp != key[i]) blanks[myid]--;
```

where `num` is local to the processor, `blanks` is a shared variable, and `myid` holds the local processor identification number. Since the last statement of this code modifies `blanks[myid]`, the second `m_sync` is necessary even though only one statement is executed after the previous `m_sync`. A user might expect that a single statement should be executed by all processors in the same amount of time, especially one that directly follows a synchronization point. This is very often not the case, however, as our debugging experiences have made clear. For example, without the second `m_sync`, other S81 users may preempt some but not all executions of the assignment to `num` so that the value assigned to `num` in the delayed processors is incorrect.

The full potential of the machine's MIMD nature is realized in the local merging phase.

Once all the local pointers have been computed, each processor can perform its own separate in-place merge on data that will not be accessed by other processors. Because processors that contain breakers must execute up to two local merges, one final `m_sync` point is used to ensure all processors have finished before the merge is completed.

Our experimentations brought forth some unexpected results. We illustrate with the following anecdote. Initial runs of the Sequent code revealed that the parallel version of the merge was curiously slow, even slower than the sequential version no matter how many processors were used. Since $O(1)$ extra storage per processor is sufficient in theory, the preliminary coding had employed only one extra memory location to be used for swapping. But six is also $O(1)$, and with just a half dozen extra swapping locations and 1/6 the number of synchronizations it became routine to get parallel execution times that were respectable.

3.2.2 Computational Experience on the S81

The architecture and programming of the Sequent Symmetry allows for many different variations in parameter values under which to test the parallel merge algorithm. Due to the available memory and program size, a maximum of 2^{21} (2,097,152) integer (4 byte) keys could be merged.

Table 3.1 shows the total amount of time spent (in seconds) by the parallel merge for a select number of processors over a number of data sizes. The times listed include the time for synchronizations (which are frequently ignored in the literature) and represent average times taken from five different, randomly generated data sets. Outside activity on the target machine was non-existent or at a minimum when the test runs were timed. The 32-bit

microsecond counter usclk was used to time the test runs.

Table 3.1: TOTAL S81 EXECUTION TIME

Number of Processors	Total Number of Records Merged						
	32768	65536	131072	262144	524288	1048576	2097152
2	1.11	2.23	4.16	8.78	17.59	35.43	71.75
4	0.76	1.40	2.80	5.30	11.13	20.63	40.26
8	0.54	0.94	1.67	3.25	6.25	12.68	24.58
16	0.62	0.87	1.37	2.36	4.75	8.44	16.68

Table 3.2 shows the average amount of time spent on each step using 16 processors with 2097152 elements to be merged. This timing was done to determine how each step was performing in relation to the others. Those steps that were taking the most time could then be looked at more closely for bottlenecks and coding inefficiencies.

Table 3.2: SAMPLE S81 STEP TIMES

Task	Execution Time (in seconds)	Percent of Total Time
Step 1 (Sort by tails)	4.49	27
Step 2 (Locate breakers)	0.00	0
Step 3 (Distribution Table computation)	1.04	6
Step 4 (Block rotation and Data movement)	7.77	47
Step 5 (Local merge)	2.47	15
Overhead (function calls, etc.)	0.91	5
Total	16.68	100

The graph in Figure 3.1 shows the relation of the overall total execution time on a differing number of keys on 16 processors.

The execution time of the parallel merge exhibited very predictable behavior on the S81. A doubling of the data size resulted in a doubling of the execution time for a fixed number

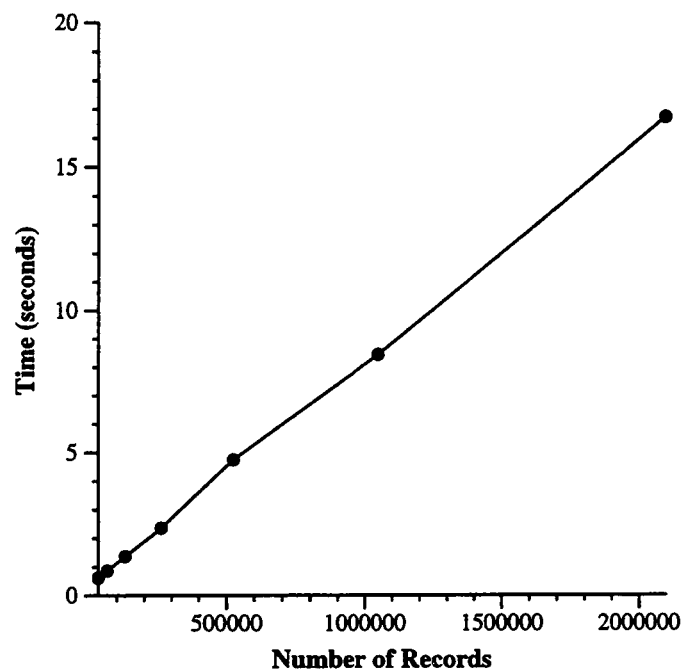


Figure 3.1: S81 TIMING RESULTS (16 PROCESSORS)

of processors. While we were satisfied with this result overall, we were somewhat surprised at the low efficiency synchronization forces on the computation.

3.3 Extension to the SIMD Model

3.3.1 C* Code Conversion

C*, the Thinking Machines Corporation data parallel version of the C language [56], was used to implement the parallel merge on the CM-2. In C*, data spread across the CM processors are placed in *parallel variables*. Each parallel variable has a *shape* that is determined by the dimensionality of the data and the number of processing elements (PEs) in each dimension.

Though variables in more than one shape can be declared, a C* program generally works with only one shape at a time (the *current* shape). To operate on a particular parallel variable, the shape of that variable must be made the current shape. The parallel variable holding the data to be merged is one-dimensional and larger than the number of processors at hand. (If the current shape holds more data elements than processors are available, then multiple virtual processors are simulated at each real processor.) Additional local variables to be used in each processor were declared to be of shape *physical*, which is the intrinsic shape that allocates a single element of the parallel variable to each physical processor. By using *physical* as the current shape and assuming the data for merging was spread across the memory in the obvious way, we were able to implement the algorithm without using virtual processors.

The major problem in coding the MIMD implementation was the number of synchronization points that had to be included in the code and the coordination of the multiple tasks in executing the same number of barrier synchronizations at the proper times. With an SIMD machine, the synchronizations are done automatically. While this makes coding of processes that are tightly coordinated much simpler, it does impose some restrictions on the capabilities of the machine that must be overcome.

The C* language does not support parallel for-loops. In the conversion of the code from the S81 to the CM-2, all such loops needed to be "hand coded" by initializing the counter, incrementing it, and checking to see if the exit conditions had been reached before each new iteration. Similarly, any loop structure that could execute on different processors with differing iteration counts had to be handled in an analogous way. Crafting a method for

simulating different length loops within the SIMD framework was tedious, yet straightforward.

Although each processor in the CM-2 must execute the same instruction that all other processors are executing, it is possible to specify that only selected processors execute statements while all others remain idle. A physical boolean parallel variable is declared and each element used to designate whether or not the loop on the corresponding processor has finished its iterations. Only those processors that have not finished are chosen to continue executing the bodies of their loops. After every iteration, each active processor checks to see if it has completed the loop executions. If so, the flag is reset to false and the processor becomes idle. When all flags are false, every processor proceeds to the next program step. The checking of all flags is performed using a boolean OR global reduction operator (`!=`) illustrated below.

For example, the following MIMD parallel code

```
i = j;
while (A[i] <= temp) {
    if (B[k] > A[i]) k--;
    i++;
}
```

would be coded as

```
i = j;
more = ([i]A <= temp);
while (!= more)
    where (more) {
        if ([k]B > [i]A) k--;
        i++;
        more = ([i]A <= temp);
    }
```

on the CM-2 where `more` is the boolean parallel variable used to indicate which processors

have more iterations to complete. (Note the reversal of the variable-index order for parallel variables used in C*.) Since each element of a parallel variable is thought to be assigned to a different processor, accessing elements that are not held by the requesting processor must be indexed. This indexing is placed before the variable name to distinguish it from the usual array indexing. (This left-indexing is executed as an indirect memory addressing into the shared memory, not an array indexing operation.) The **where** statement is the C* equivalent of a parallel **if-then-else**. Only those processors that evaluate the conditional statement as **true** are allowed to execute the **where** clause statements. Following this, if there is an **else** clause, those processors that had been idle for the **where** clause are then activated to execute.

The C* function library includes routines that can perform computation on parallel variables and transmit data at the same time. The **scan** operation is the most generic of such operations and is easily adapted to perform other, similar tasks. Parallel prefix operations are computed with the combination of the **scan** operator and the specification of how to combine elements of the parallel variable. Possible combiner functions are **MIN**, **ADD**, **MULTIPLY** and logical **AND**. The direction of a scan operation is determined by the user. The *upward* direction scans from the lowest processor index to highest, and the *downward* scans from highest to lowest.

The broadcasting of breaker values in the second step of the merge is performed by the scan facilities on the CM-2. To accomplish a scan that will only broadcast values from a single processor to all processors in the series, but not to others outside the series, each broadcasting processor assigns the broadcast value to the appropriate parallel variable. (The

broadcast values are all local indices and are monotonically increasing, as are the processor identification numbers.) All other processors place a zero value in this same parallel variable. The scan then performs a MAX scan in the upward direction. Since the values to be broadcast are greater than zero, they replace all the zero entries. The broadcast segment halts when the next non-zero element is reached within the parallel variable. The same broadcast can be done in a downward direction by placing large values instead of zeroes and performing a MIN scan. C* is defined to provide this type of segmented scan without having to rely on the values of the data to delimit the segments. Unfortunately, this feature was not implemented on the CM-2 we had available for our tests.

The use of a parallel scan as a broadcasting function led to an improvement in how the broadcasting of data could be done in the S81 code. Rather than the complicated algorithm originally proposed in [28], appropriate scan and one-to-many broadcast operations were hand-coded using pointer jumping techniques [65, 14]. This approach sped up processing of the broadcast portions of the code by about 10%. Although this savings is insignificant in the overall execution time, the resultant code is considerably simpler.

Another form of scan operations are the reduction operators in C*. These operators reduce a parallel variable to a scalar value by combining the values of the parallel variable according to the given associative operation. Summation and bitwise logical operations, along with MIN and MAX, are among the possible combining operations of the reduction operators.

The sorting of tail keys in the first step of the merge algorithm is performed by another specialized function, `rank`. The `rank` function returns a parallel variable of the same size as

the given parallel variable. This returned variable holds the rank of the data values in the corresponding positions. This variable is used as an index to swap the blocks.

3.3.2 Computational Experience on the CM-2

The CM-2 was not as flexible in allowing variation in run-time parameters as the S81. Once the size of the data was determined, the number of processors was the only other parameter that could be adjusted. The choices allowed on the test hardware were 8192 or 16384. Irrespective of the number of processors, we discovered that the timing results were remarkably well correlated to the number of keys allocated per processor. That is, the execution times for merging 32768 total keys on 8192 processors was the same as merging twice as many keys on 16384 processors.

With 16384 processors, memory space allowed a maximum of 2^{21} (2,097,152) integer (4 byte) keys, matching nicely with the S81 limit. This bound gives a ratio of 128 keys per processor, and was the upper limit on this ratio no matter how many processors were employed. (In other experiments it was found that programs with fewer statements could compute with larger data sets.)

Table 3.3 lists the total amount of execution time (in seconds) spent on the parallel merge using 16384 processors with differing data set sizes. The results shown are based on the average of five random data sets and were generated on a 16384 processor CM-2 making full utilization of the machine and its connected resources. These results were obtained through the standard CM-2 timing routines of `CM_start_timer(1)` and `CM_stop_timer(1)`. Real (wall clock) time, CM time, front end virtual time, and percentage of CM and front

end utilization are reported by these routines. We use the front end virtual time because it measures the time used by the program in execution on both the front end machine and the CM-2 processors. Table 3.4 shows the time taken by each individual step of the computations from merging 2097152 keys with 16384 processors. Figure 3.2 gives a graphical representation of the CM-2 execution times.

Table 3.3: TOTAL CM-2 EXECUTION TIME

Number of Processors	Total Number of Records Merged						
	32768	65536	131072	262144	524288	1048576	2097152
16384	0.45	0.51	0.79	1.66	5.23	20.21	74.41

Table 3.4: SAMPLE CM-2 STEP TIMES

Task	Execution Time (in seconds)	Percent of Total Time
Step 1 (Sort by tails)	1.81	2
Step 2 (Locate breakers)	0.07	0
Step 3 (Distribution Table computation)	1.85	3
Step 4 (Block rotation and Data movement)	6.35	8
Step 5 (Local merge)	63.87	86
Overhead (function calls, etc.)	0.46	1
Total	74.41	100

3.3.3 MPL Code Conversion

MPL [45] is the C-like programming language for the MP-2. MPL is more like C than C* is. Common structured programming constructs `if-then-else`, `while`- and `for`-loops are all available. Where the programmer must manually control the set of CM-2 processors during execution of loops with independent iteration lengths, this is automatically handled by

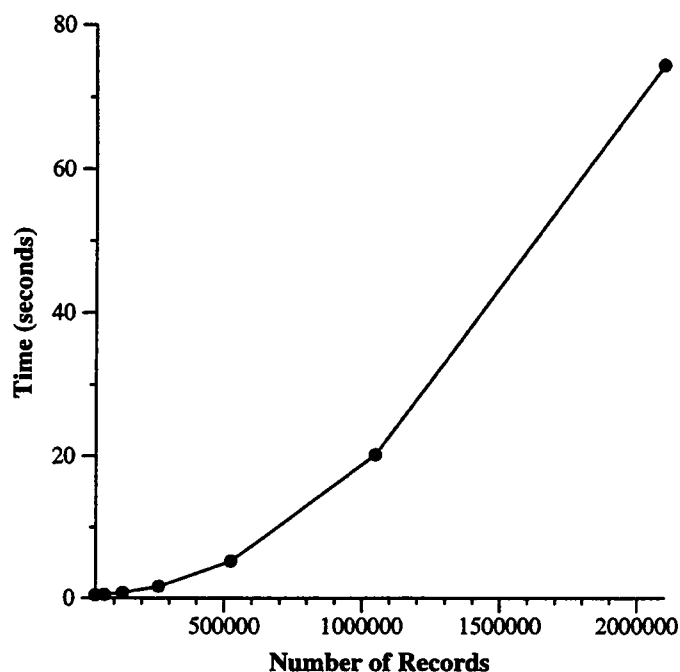


Figure 3.2: CM-2 TIMING RESULTS (16384 PROCESSORS)

the MPL compiler and the MP-2 controller and processors. This feature alone would make code conversions from MIMD machines to MPL much more straightforward. Unfortunately, MPL does not contain the wealth of global `scan` and `reduction` functions found in C*. We hand-coded these operations following the modified codes written for the S81.

Programming on the CM-2 assumes a shared memory paradigm though the memory is local to processing nodes within the hardware. The MP-2 is programmed under a distributed memory scheme. Parallel variables declared as *plural* in MPL are duplicated across all processors. The data to be merged is distributed among separate arrays held by each processor. Individual records not held in local memory are accessed through the router by knowing the *id* number of the owning processor and the offset of the desired record within that processor's local array. The EREW PRAM design of the merge algorithm ensures that

Table 3.5: TOTAL MP-2 EXECUTION TIME

Number of Processors	Total Number of Records Merged						
	32768	65536	131072	262144	524288	1048576	2097152
4096	0.06	0.12	0.20	0.41	0.86	1.95	4.25

no two processors will attempt to access the same record at the same time.

3.3.4 Computational Experiences on the MP-2

Table 3.5 shows the total execution time of the G-L merge on a MasPar MP-2 with 4096 processors. While the memory space available on the machine can accommodate over 30 million 4-byte integer keys, we have only included run times for those data sizes that were reported for the S81 and CM-2 earlier. The timings are an average of five randomly generated data sets using the MPL functions `dpuStartTimer()` and `dpuEndTimer()`. These timers count wall clock time that is used while the program is executing.

The graph in Figure 3.3 includes many more data sets than are shown in Table 3.5. From this we conclude that the MP-2 does not suffer from the same memory management problems as the CM-2 and the G-L merge scales well on this machine.

When the execution time is broken down by algorithmic step, some interesting results are revealed. Table 3.6 shows the time taken for each step on a 4096 processor MP-2 merging a total of 2097152 integers. The reader will note that there was no unaccounted overhead time found in the step timing of the MP-2.

The MP-2 percentage values follow the same general pattern as those for the CM-2 (Table 3.4). As with the CM-2, the major portion of the execution time is spent in the final

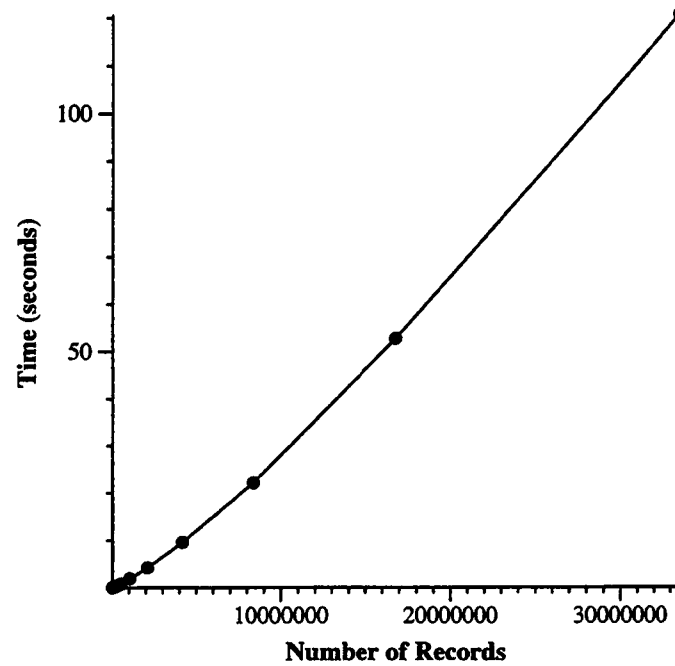


Figure 3.3: MP-2 TIMING RESULTS (4096 PROCESSORS)

Table 3.6: SAMPLE MP-2 STEP TIMES

Task	Execution Time (in seconds)	Percent of Total Time
Step 1 (Sort by tails)	0.40	10
Step 2 (Locate breakers)	0.02	0
Step 3 (Distribution Table computation)	0.22	5
Step 4 (Block rotation and Data movement)	0.65	15
Step 5 (Local merge)	2.97	70
Total	4.25	100

local merge step, though by a smaller proportion. The larger percentages of time spent in the data movement steps (steps 1 and 4) indicate to us that the router, like the bus used on the S81, could be a potential bottleneck for irregular communication patterns often found in non-numeric computations.

3.4 Interpretation of Results

3.4.1 Initial Remarks

Conversions from one architecture to another are unattractive if one must program with low-level tools. Thus all our methods were implemented in high-level languages accessible to all users. Dramatically better timing results would have been possible if certain sections of our algorithms were microcoded (as was done, for example, in [11]). But this was not our goal. We sought instead to identify algorithmic bottlenecks of the transformation process.

More interesting than any direct comparison of execution times between the two machines is a comparison of the strengths and weaknesses of both models by how each performed on the various steps of the algorithm.

On the S81, the execution times roughly double as the amount of data doubles for a fixed number of processors, while the execution times do not decrease by half when the number of processors is doubled and the input size is fixed. This suggests that certain features of this machine (discussed in § 3.4.4) do not scale very well with the addition of extra processors for algorithms such as this one.

The parallel merge has quite the opposite effect on the CM-2. Twice as many processors

take half as long on the same data sets. Yet, when the number of processors is held constant, a doubling of the data resulted in a quadrupling or more of the execution time. Uncovering causes of this behavior (discussed in § 3.4.2) has been illuminating.

Execution times on the MP-2 doubled as the data set size doubled for relatively small data sets. As the number of records increased, the execution time would increase by slightly more than a factor of 2 for each doubling of the data size. This was nowhere near the quadratic growth of the CM-2. We offer an explanation of this behavior in § 3.4.3.

3.4.2 Data Locality and Cache Effects

Data locality is often claimed to be one of the most important factors in (sequential and) parallel computations. Indeed, this is seen quite well in the step timings (Table 3.2) on the S81. The first and fourth steps move large numbers of keys from all over memory to different memory locations. This can swamp the single data bus. The processor cache is not able to alleviate this problem even though the movement of data is done in an orderly, regular manner that could be useful in a cache prefetch scheme. Since it has little or no effect on the execution of this step the loading of the cache may slow down these portions of the code.

One might assume that the CM-2 would also suffer from this problem since data must be moved between processors that are not necessarily located in close proximity to each other. Folk wisdom holds that communication should be kept to an absolute minimum, and should be done over the smallest proximity of processors whenever possible. Nevertheless, the CM-2 routing network is able to shuffle large amounts of data between processors without much danger of running into the collision possibilities inherent in the single data bus on the

S81. Moreover, the relatively low number of keys assigned per processor significantly reduces the number of data transfer instructions that need to be executed. (The largest data ratio achievable on the CM-2 was 128 keys per processor while the largest ratio on the S81 was 524288 keys per processor.)

On the S81, bus and memory contention can be reduced by performing the local merge on data held in the cache. The three intermediate steps of the algorithm require data transfer and cooperation between processors that are within the same series pair. Since data located between breaker records will remain between those breaker positions after the final merge completes, the more breakers contained in the data the tighter the data locality will be within these three steps. The large number of synchronizations required in the fourth step contribute to the large amount of time spent there.

The CM-2 performs well in these intermediate steps by being able to perform intrachip communication. Since processors are grouped 16 to a node, when the number of breaker records is very high much of the required data transfers can take place between processors on the same chip and the same memory module.

We had not anticipated the small percentage of time the Symmetry required for the local merge. In this step, all data can be manipulated within a processor's cache. As the algorithm rolls through the data, the in-place merge has a more predictable memory access pattern that results in very high cache hit ratios. It was similarly surprising that the local merge step took the most time on the CM-2. One explanation proposed was the serialization imposed by the SIMD model (more on this later). Another possibility was the contention of all 16 processors

on the board for the same memory. By eliminating all but one processor performing the local merge, this last step of the algorithm exhibited a 20% average decrease in execution time from previous timings with all 16 processing elements (PEs). However, the behavior of more than doubling the execution time when doubling the data size remained. (The MP-2, built and programmed with distributed memory, does not have any memory contention problems unless the same remote location is requested by more than one processor.)

After conducting many separate experiments to trace the cause of this problem, it was concluded that the source of the problem was based on memory access on the CM-2. One such experiment had each processor loop through n/p iterations performing a swap of two local locations from a large parallel variable. As the size of the parallel variable doubled, the runtime of this simple code also increased fourfold or more.

Several consultations with experts at Thinking Machines uncovered that some access patterns are highly non-optimal. We were advised that our only alternative was to augment the C* code with memory access functions available in CM-2 assembler (PARIS). Given our focus, we of course avoided such a low-level fix. Observe that this is exactly the sort of result we sought to uncover: the state of the art is still that one cannot reasonably expect straightforward, high-level transformation from one architecture to another without degradation in performance.

3.4.3 Single Instruction Limitations

As mentioned earlier, the final algorithm step comprises two distinct merge phases due to the disconnected nature of data held in processors that contain breakers. The SIMD paradigm

requires that these two merges be executed in serial, one following the other, across all processors with breaker records. (Those processors without breakers remain idle during the execution of the second merge.) Additionally, within the local merge itself, there are two distinct merge routines: one for those processors that contain a very small list to be merged and the in-place merge for all others. Due to the lockstep execution mode of the CM-2 and MP-2, these must be run in serial across all the processors participating in the merge. Thus, there are in effect 4 separate merges that must be executed in the final step of the algorithm with no overlap possible.

When the data sets are very small, it is likely that all processors will contain a short list. If there are no processors holding two lists of sufficient size, the full in-place merge is not executed—rather, a buffer merge is performed—and the total execution time is kept small. As the total number of items grows, the likelihood of all processors having small lists decreases while there is still a good chance that at least one processor will have a short list (especially those processors that contain breakers). This extra processing accounts for the slight increase in the ratio of execution time to data size above 2 on the MP-2 as the data set size increases.

Two other inherent SIMD serializations occur when using the C* **where-else** construct and arrays. The execution of the **where** clause is followed by the execution of the **else** clause by the appropriate processors. Again, no overlapping of separate executions is possible.

Access to array elements with a variable index is serialized by the front end processor. This is a very important architectural concern. A master index is iterated through all possible

array index values and only those processors that have a matching index value are allowed to execute. Thus, in principle, an operation that appears to be $O(1)$ is really $O(n)$. In separate tests, implementations of simple merging and sorting routines using local arrays proved to be an order of magnitude faster for the small number of keys used by each processor. When testing a version of the parallel merge code that used arrays at each processor instead of a single parallel variable, memory access errors developed during the displacement table building step. Only specialized array handling routines can correct this problem, and we again shunned those because they are not widely distributed for general use.

The MPL `if-then-else` construct suffers from the same serial execution of `then` and `else` clauses. However, the use of arrays on the MP-2 is not a bottleneck. Each processor is able to use a local variable for access into local arrays. In fact, through the communication router network, processors may use a local variable to access an element from a non-local array.

3.4.4 Task Synchronizations

Because of the limited types of S81 runtime data available, the precise amount of time taken by barrier synchronization cannot easily be determined. (The best estimate that we were able to find was obtained by running the merge code with only the `m_sync` calls intact.) The number of loop iterations for data movement steps was approximated based on the size of data and number of processors being used. Times from these experiments ranged from 10% to 25% of the total execution times for the full merge with corresponding amounts of data and corresponding numbers of processors. This method of timing was not able to take into

account the actual overhead time that processor interactions inflict on execution time. Thus, we assume that the sync times can be much higher in practice, especially as more processors are added.

While the execution time on a fixed number of processors doubled as the merge data size was doubled, doubling the number of processors for a fixed data set size did not cut the execution time in half. The impact of synchronization overhead is most apparent when comparing the execution times of Steps 4 and 5 from Table 3.2. Both execute approximately the same number of data moves, yet the fifth step executes in $1/3$ the time of the fourth step. There is no synchronization required in the local merge, while the data movement between processors in the fourth step is heavily synchronized.

Although the CM-2 and MP-2 do not require explicit processor synchronization, an overhead is charged to the user for the SIMD lockstep mode of operation. There appears to be no accurate way of determining this overhead cost.

3.5 Some Lessons Learned

The transformation of an algorithm from the Sequent Symmetry S81 to the Thinking Machines CM-2 and MasPar MP-2 has helped illustrate that performance depends heavily on how these resources are handled by each machine. In particular:

- Large-scale data movement is hampered by the single data bus design of the S81 while the router networks of the CM-2 and MP-2 are able to move data between different processors without much apparent difficulty.

- Memory management on the CM-2 can be primitive, despite other advanced architectural features of the machine.
- Explicit user-placed synchronizations needed in the MIMD model and the inability to process different parallel tasks under the SIMD model are major drawbacks for complex non-numeric algorithms such as ours.

3.5.1 Architectural Performance

The wealth of built-in functions that are designed with efficient data movement in mind is a large advantage of the CM-2 over the S81. The `scan` function and its many incarnations alleviate a large burden from the programmer and allow a somewhat higher level of programming to be done on the CM-2. Having to emulate some of these same functions on the S81 led us to appreciate the amount of effort that was saved through their use. The mesh communication routines on the MP-2 are much faster than moving data through the router network. It seems unlikely that non-numeric algorithms would be able to take advantage of this restricted form of processor interaction.

As already noted, a significant drawback of the S81 is its synchronization overhead. The less tasks must cooperate with each other, the better suited the algorithm is for MIMD execution. The transferral of data between processors will necessitate some amount of synchronization. Barrier synchronization is, perhaps, the wrong type of synchronization primitive to be used in non-numeric parallel algorithms. A better synchronization scheme would be to have only those processors that are cooperating with each other synchronize together. Alas,

programming and debugging with such synchronization could easily become a monumental task. Distributed-memory MIMD machines can employ their message passing facilities to synchronize only those tasks that need to cooperate closely as well as share data between processors. The connection network of such a system would also be able to move large amounts of data more efficiently between processors. We shall examine the implementation of the G-L merge and its performance on distributed-memory architectures in the next chapter.

When algorithms call for very close cooperation between processes, SIMD machines offer synchronized execution. Without having to coordinate multiple synchronization points over tens of thousands of processors, the CM-2 is relatively simple to program. Machines like the CM-2 may not, however, perform well on array accesses, *if-then-else* statements and loops of differing iteration lengths. This is regrettable since non-numeric algorithms tend to contain these programming structures in abundance.

Chapter 4

Shared Memory to Distributed Memory Conversion

4.1 Introduction

In the previous chapter, we discussed the conversion of the G-L merge from an MIMD architecture to an SIMD one. In this chapter we shall examine the conversion of the same algorithm from a shared memory machine to one utilizing distributed memory, specifically from the Sequent Symmetry S81 to the Thinking Machines CM-5. The G-L merge algorithm was initially developed on the EREW PRAM model. The shared memory implementation of the algorithm followed these EREW restrictions. Algorithms on distributed memory models are inherently EREW since processors may access only their own memory space directly. To access data held in other memories requires that messages be passed between processors.

Several programming challenges were encountered in our effort to implement an algorithm designed for shared memory execution onto a distributed memory system. However, we were also able to uncover and explore alternate methods of performing tasks that were

not available or not immediately suggested by the shared memory programming model.

We know of no previous research into the translation of shared memory programs onto distributed memory machines. In practice, efficient shared memory programs for matrix and related linear algebra computations are those that keep as many of the available processors doing useful computations as possible. The “pool of tasks” technique of dynamic scheduling is a popular methodology used in shared memory programming. On distributed memory systems, the efficiency of numeric algorithms depends heavily on the distribution of data among the processor memories and the interconnection network between processors.

Shared and distributed memory paradigms would appear to have minimal overlap. Implementing shared memory numeric algorithms on distributed memory machines required programmers to redesign codes to take advantage of or avoid the pitfalls of the interconnection network. Also, it was found that the shared memory algorithms did not scale well on machines equipped with many more processors [35]. The resulting codes take little or nothing from their shared memory predecessors. Even sources that describe both shared and distributed memory implementations of the same matrix algorithms support this dichotomy [23, 24, 29]. The two paradigms are treated as separate; no direct conversion of codes from one model to the other is offered. Zimmermann and Wölf [66] give a method of translating a specific class of shared memory PRAM algorithms to the distributed memory LogP model [15]. This class is made up of those algorithms that have a predictable communication behavior. In [41], the (shared memory) PRAM model is used to illustrate the practicality of a parallel solution for problems while algorithmic implementations are almost exclusively

presented for distributed memory models.

Recently, the shared memory model of computation has regained popularity. Many proposed systems (e.g., [43, 49]) have sought to simulate a shared memory programming interface on distributed architectures through software. Besides being able to present a single programming environment across different machines, the major goal of these *distributed shared memory* systems is to automate the efficient allocation of data and processing across the distributed processors and memory. Non-local memory references are handled automatically during program execution alleviating the programmer from having to explicitly move data from one processor to another via messages. High Performance Fortran (HPF) [30], when implemented on distributed memory machines, must do similar automatic message passing to access shared memory allocated in a distributed fashion.

4.2 A Simple Technique

Shared memory computations using statically allocated tasks have predetermined divisions of data among processors. For example, sub-blocks of files (for list processing) or subsets of vertices (for graph algorithms) are assigned to be under the control of distinct processors. In some cases, the division of data is little more than a device to spread the amount of computation equitably across processors. Access to data not assigned to a processor is easily accomplished through the shared memory. The G-L merge is such a statically allocated, shared memory algorithm.

Given a statically allocated, shared memory algorithm, we first translate the shared

memory code to a compatible language version for the distributed memory machine. All points within the shared memory code in which data is being shared between processors are identified. A communication sequence replaces the direct access of data. In what follows, we borrow ideas from flow dependence for vectorizing compilers and pipeline computation [40, 38].

We define the *read set* and *write set* of a statement executed by processor P_i as follows:

Definition 4.1 *For statement S executed on processor P_i , the read set, $R_i(S)$, is the set of memory locations used in the statement, but not modified.*

Definition 4.2 *For statement S executed on processor P_i , the write set, $W_i(S)$, is the set of memory locations modified in the statement.*

These definitions are extended to a block of two or more contiguous statements in an obvious way.

Mutual exclusion of data access can be enforced by augmenting a shared memory code with synchronization points. Such a synchronization point is required if two statement blocks, with the potential of simultaneous execution, interact such that either (1) the read set of one block overlaps the write set of the other or (2) the write sets of two blocks overlap. (For algorithms designed with Exclusive-Write restrictions, this latter case should not need to be considered.) These synchronization points can be one of several mutual exclusion devices for coordinating two processes (see [8] for examples) or simply a barrier synchronization if the entire set of processors may have intersecting memory access requirements. This ensures

that the proper values are contained in the shared memory locations before they are read and/or updated by another processor.

Definition 4.3 *The set of memory locations statically allocated to processor P_i is the hold set, $H(P_i)$.*

The above definition is applicable to both shared and distributed memory models. For our purposes we shall consider the hold set to be those memory locations from the shared memory algorithm statically allocated to distributed processors.

If the read set or write set of a statement from the translated shared memory code executing on distributed processor P_i has a non-empty intersection with the hold set of another processor $j \neq i$, $H(P_j)$, there is a need for communication. The actual interaction between processors depends upon the nature of the data that is to be used by P_i .

If $W_i(S) \cap H(P_j) \neq \emptyset$, then statement(s) S , when executed on P_i , must be modified or supplemented to send a message to P_j that will update data held by P_j . The algorithm is also modified in such a way that P_j will receive the updated values at an appropriate time. In the case where $R_i(S) \cap H(P_j) \neq \emptyset$, P_i must receive data held by P_j , there are two possible cases of communication sequences:

Case 1. If the data values to be communicated can be determined directly from the algorithm (e.g., a specific variable, the fifth element of an array, etc.) a message containing the data from the holding processor, P_j , to the receiving processor, P_i , is all that is needed.

Case 2. If the data values required are determined by values or conditions local to P_i , a request message must first be sent. P_i constructs and sends a message requesting the desired data, this message is received and processed by P_j , and the requested data is composed into a message and sent back to P_i .

The second case sets up a consumer/producer relationship between processors. Depending upon the data allocation and movement requirements of the processors and the algorithm, there may be disjoint sets of consumer and producer processors or there may be processors that act both as consumer and producer processes. In this latter case, the implementation must take into account the receipt and processing of data requests and fulfillment of requests on the same processor. Such algorithmic modifications must ensure a balance is achieved between the processing of data received and the satisfaction of requests received for maximum parallel execution. That is, processors acting as both consumer and producer cannot focus on a single type of message processing to the exclusion of the other. Such exclusion will certainly result in a delay of other processors and could deadlock the computation. We shall see such a case of processors that act as both consumer and producer during the displacement table computation of the G-L merge.

It is possible, with some extra thought and knowledge of the algorithm, to reduce the number of total messages dictated by our statement-by-statement analysis. For example, if more than one value is needed from $H(P_j)$ within a block of statements, a compound message that transfers all required data items could be sent to P_i (dependent on the message-passing capabilities of the target system). Care must be exercised to ensure that processor P_j will

not update any of the data items that are sent before processor P_i will have made use of their values. For SPMD programs, a single communication block will suffice if it can be guaranteed that each sending processor has a unique receiving processor for the data being sent.

Different combinations of algorithms and target architectures will bring up special cases that must be addressed individually. This section does not seek to be an exhaustive treatment of converting shared memory codes to distributed memory machines. Rather, it is intended to give the reader a starting point from which to begin the transfer process. Efficiency issues regarding interconnection network, cache utilization and other architecture related issues are beyond our scope. Below we examine the conversion process of the G-L merge in more detail.

4.3 Conversion Details

4.3.1 CMMD Advantages

CMMD [57] is the message-passing library for the Thinking Machines CM-5. Functions for sending and receiving messages (both synchronously and asynchronously), global broadcast, synchronization and various auxiliary information functions are provided. The CMMD library also contains specialized global functions for performing parallel scan, reduction and concatenation operations across all processors.

As before, it was thought that global broadcast routines, not available on shared memory machines, would be particularly useful in our conversion. However, after the SIMD implementation on the Thinking Machines CM-2 using the `scan` operation to perform a segmented

broadcast, we found that using the CMMD scan routine in a similar manner would be a better implementation.

The one-to-many broadcast operation was used to inform all processors of which processor was "last" (i.e., the rightmost second series processor) in the computation. Once breaker positions are found, all records following the rightmost breaker are in final sorted order. Any processors to the right of the "last" processor have completed processing their data and interacting with other processors. Due to the nature of the CMMD scan operation, each processor initially included in the program execution must perform a scan for the operation to successfully complete. Thus, all processors found to be to the right of the "last" processor are directed into a portion of the code that contains only an appropriate number of calls to the global communication routines with dummy values that will not adversely affect the results for other processors continuing execution.

This early exclusion of processors is handled on the S81 by preventing such processors from executing blocks of code. Instead, these processors are directed to execute a number of `m_sync()` calls that correspond to the synchronization points found in the skipped block. In this way, changes in the number of synchronization points were localized and much less likely to cause execution to deadlock from mismatched `m_sync()` calls. On the CM-2, such processor exclusion was done by masking out those processors that had been found to have completed processing their assigned data.

4.3.2 Examples

To determine if a data block contains a breaker, the processor assigned to that block must have the tail key of the preceding processor. On the S81, this was done with the statement

```
if (myid != 0) temp = A[low - 1];
```

where `myid`, `temp` and `low` are local (private) to each processor and the `A` array is the shared array of records. The value in `low` is the index of the head within the block of data assigned to the processor which makes `A[low - 1]` the tail of the left neighbor processor. P_0 is excluded from retrieving this tail record since it has no processor to its left.

Using our simple technique, the message passing code on the CM-5 has every processor participating in some communication. The communication corresponds to Case 1 from § 4.2 since it is the algorithm that determines what data item (the tail of the left block) is needed. Each processor must send its tail to the right and receive the tail from the left. Exceptions to this message passing are processors P_{p-1} and P_0 , which have no right neighbor or left neighbor, respectively. The CMMD code is

```
if (myid < num_procs-1)
    CMMD_send(myid+1,tag,&A[bsize-1],sizeof(int));
if (myid != 0) {
    CMMD_receive(myid-1,tag,&buffer[0],sizeof(int));
    temp = buffer[0];
}
```

where `A[bsize-1]` is the tail in local memory for each processor and `num_procs` is the number of processors participating in the computation. The parameters for the `CMMD_send` and `CMMD_receive` are processor *id* number to send to/receive from, message tag, pointer to message buffer and number of bytes in message.

The computation of the index of the second series processor at each associated first series processor presented a more complex conversion scenario. By our simple technique, this code segment would have fallen under the Case 2 consumer/producer framework. With a little insight we were able to avoid such a complicated coding.

On the S81 the index of the breaker is broadcast to processors to the left and stored in the shared array `lbearer` at the element indexed by the processor's `id` number (`myid`). The quotient of the integer division of `lbearer` by the size of the data blocks for each processor (`bsize`) equals the processor index of the second series processor. (Note: though the local record indices range from zero to `hi` on distributed memory processors, the value of `lbearer` and any other local index broadcast to other processors will be the index value such records would have if the list of records was contained within a single, global array.) Unfortunately, as a side effect of the breaker index broadcast algorithm, second series processors (those with non-zero `breaker`) with first series records following their local breaker do not receive the broadcast value. These processors can access the `lbearer` of their right neighbor and compute the second series processor index. The S81 code for this computation (which would immediately follow the code for broadcasting `lbearer`) is below.

```

if (breaker[myid] == 0) {
    lastblock = lbreaker[myid] / bsize;
    m_sync();
}
else {
    m_sync();
    if (myid != nprocs-1) lastblock = lbreaker[myid+1] / bsize;
}
m_sync();

```

Since only those processors whose data blocks have breakers need to communicate, our technique would recommend that these processors send a request to the processor on their right. All processors, however, must check for the receipt of such a request. Although there are functions under CMMD that can determine if a message has been received, there is no simple method to ascertain that no message is going to be received.

For the conversion of the above S81 code segment, all processors (with appropriate restrictions on the first and last processors) send and receive an lbreaker value. After the values have been received, processors decide if the original local value or the received value is needed for computation. In this way, a potentially complicated communication scheme is simplified. The corresponding CMMD code is below.

```

if (myid == num_procs-1)
    CMMD_send(myid-1,tag,&lbreaker,sizeof(int));
else if (myid == 0)
    CMMD_receive(1,tag,&buffer[0],sizeof(int));
else
    CMMD_send_and_receive(myid+1,tag,&buffer[0],sizeof(int),
                          myid-1,tag,&lbreaker,sizeof(int));
if (breaker == 0)
    lastblock = lbreaker/bsize;
else
    if (myid <= last) lastblock = buffer[0]/bsize;

```

4.3.3 Displacement Table Computation

The most radical change to the shared memory version of the G-L merge algorithm was in the displacement table computation step. The displacement table entry for each first series block is the number of records that must be traded between itself and the associated second series block to place all records in their final block destination.

The EREW PRAM model algorithm calculates this table entry by finding the sorted position of each first series block's tail among the second series block's records and then conducting a linear search of the second series keys from that position. Since each first series block will locate a different position within the second series, the linear search is assured EREW access.

The more intricate phased merge algorithm was chosen in lieu of a binary search to preserve the EREW nature of the shared memory algorithm. Since EREW restrictions are inherently preserved by message passing models, our CM-5 implementation is able to use a modified binary search to find a first series block tail's sorted position within the second series data.

Before the search can begin, each second series processor does a segmented broadcast of the size of its second series block and the number of first series processors within the series pair. This segmented broadcast is directed at the processors holding the associated first series records and is accomplished with scan operations as detailed above.

After receiving the relevant data, first series processors compute an offset into the second series records based on their position within the series pair set. This first offset is used as the

initial "midpoint" index within the second series block. Requests for individual keys at the relevant midpoint position are sent to second series processors. Upon receiving a key from the second series processor, that key is compared against the local block's tail key. Based upon the outcome of the comparison, the next key index request is computed by determining the midpoint index between the received key and the appropriate end of the second series block. A message requesting this new midpoint key is sent to the second series processor. The parallel binary search continues in this way. After the position of the local tail key within the second series block is determined, first series processors send requests for sub-blocks of data upon which to conduct their linear search that will determine the displacement table value for the first series block.

The major difficulty in programming the above was coordinating the second series processors that contained both second and first series records. Besides receiving and handling requests from first series processors, each second series processor must make its own set of requests and searches in support of its own first series records. Message tag numbers are used to determine the type of message being sent and received and to select the appropriate handling of the message. Each message sent to a second series processor is handled in the order received and processed to completion. That is, for data requests from first series processors, the data to be sent is placed into a message and returned to the requesting processor; for data being returned as the result of a request, either a new midpoint index is computed or the block of keys is searched before a new data request is sent.

Table 4.1: TOTAL CM-5 EXECUTION TIME (32 PROCESSORS)

Number of records merged	Time (seconds)
524288	0.95
1048576	1.83
2097152	3.75
4194304	7.28
8388608	16.91
16777216	29.19
33554432	69.87
67108864	119.32
134217728	232.81

4.4 Computational Experience on the CM-5

As with the previous conversion test runs, two sorted lists of randomly generated 32-bit integers were used to conduct timing executions of the CM-5 implementation. The two same-sized lists were evenly divided between the processors on the CM-5. Table 4.1 shows execution times for the merge on various numbers of records on a 32-node CM-5. Times were measured using the `CMMD_node_timer` functions of which the elapsed time, that time the program was scheduled for execution on the system, was recorded. Times reported here are the average of five different, random test runs for each data set size. Figure 4.1 displays the data graphically. As can be seen, this MIMD implementation exhibits a linear scaling for increasing data set sizes.

Corresponding timings for the same number of keys on 16 processors of a Sequent S81 are compared with CM-5 times are shown in Table 4.2. The CM-5 execution times are over four times faster than the S81 equipped with half as many processors.

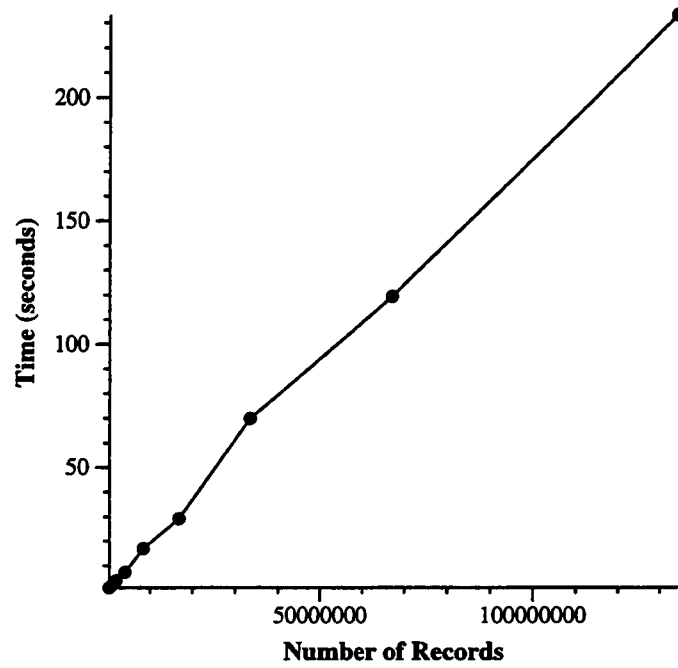


Figure 4.1: CM-5 EXECUTION TIME

Table 4.2: COMPARISON OF CM-5 AND S81 EXECUTION TIMES

Machine	Total number of records merged		
	524288	1048576	2097152
CM-5	0.945	1.831	3.752
S81	4.75	8.44	16.68

Table 4.3: CM-5 RUN TIMES WITH DIFFERING MAXIMUM MESSAGE SIZES

Number of keys merged	Time (seconds)			
	10 keys	30 keys	50 keys	100 keys
524288	0.95	0.48	0.40	0.34
1048576	1.83	0.93	0.78	0.66
2097152	3.75	1.88	1.57	1.79
4194304	7.28	3.70	3.57	2.61
8388608	16.91	8.29	6.82	6.15
16777216	29.19	14.76	13.34	12.86
33554432	69.87	34.18	27.80	27.42
67108864	119.32	59.42	50.01	49.71
134217728	232.81	118.23	102.92	99.98

4.4.1 Message Size Variations

As in the previous chapter, we found that the maximum number of items moved at a time can directly affect the execution time. To remain an in-place algorithm, the maximum size of a message must be a constant, but variations in the value of that constant can produce dramatic differences.

Table 4.3 compares run times for four different maximum message sizes (or *block size*, defined as a constant BLOCKSIZE within the code) that can be contained within a single message: 10, 30, 50 and 100 records per message. This maximum limit is used when moving records in bulk during tail sorting (Step 1), linear search within displacement table computation (Step 3) and the transfer of records between processors to set up final local merge (Step 4).

There is a point of diminishing returns on the block size. Distributed memory systems have a maximum message size set by the hardware. Any message that exceeds this bound is

sent in multiple parts.

We also ran experiments to find the total execution time for various values of BLOCKS-IZE ranging from 100 records to 3000 records in order to merge a total of 134217728 records. Although the best execution time for the single 134217728 record data set occurs at a block size of 2200 records, we chose to use a block size of 100 records (taking only an additional 8% over the execution time of 2200 records) in future distributed memory implementations of the G-L merge. This block size is small enough to be in keeping with the spirit of using a limited, constant number of auxiliary memory locations for in-place algorithms. It also appeared to be the smallest value that gave good consistent performance results over a wide range of data set sizes.

4.4.2 Timing by Step

Table 4.4 shows the average maximum amount of time spent on each step using 32 processors on the CM-5 under CMMD with 2097152 elements to be merged with a block size of 100 records. This timing was done to determine how each step was performing in relation to others as in the previous chapter.

Each of the step times is the average of the maximum time taken by a single processor (possibly different for each test run) over five randomly generated test runs. Because of the asynchronous processor execution, different processors will take differing amounts of time on the same step depending upon the data involved. Had a barrier synchronization been placed before the beginning of each step, the execution time shown in the final column of Table 4.3 would more closely reflect the "total" time computed in Table 4.4 above. While not wholly

Table 4.4: SAMPLE CM-5 STEP TIMES

Task	Execution Time (in seconds)	Percent of total time
Step 1 (Sort by tails)	0.36	15
Step 2 (Locate breakers)	0.01	0
Step 3 (Distribution Table computation)	0.47	19
Step 4 (Block rotation and Data movement)	1.19	48
Step 5 (Local merge)	0.43	18
Total	2.46	100

accurate with regards to actual execution time, Table 4.4 does indicate the trends exhibited by the steps of the computation in relation to each other.

4.5 Interpretation of Results

4.5.1 Irregular Communication

Two general techniques for improving efficiency in implementations of numeric computations on distributed memory machines are (1) partitioning tasks and data to match the resident network interconnections and (2) overlapping of communication with computation.

It is highly unlikely that a pattern of communication within a given non-numeric computation on a particular data set will match the network topology. Even if such a data set were to be found, it would not be representative of the vast majority of all applications. Also, overlapping communication and computation is virtually impossible for non-numeric algorithms, especially those based on data movement. In the case of the G-L merge, the identity of data needed for further processing is often unknown until just before it is required. Furthermore,

there are relatively few computations found between communications.

While the irregular communication patterns of the G-L merge may not be suited to perform optimally on distributed memory machines with fixed interconnection networks, there are advantages of such implementations. Comparing the step execution time percentages from Table 3.2 and Table 4.4 reveals that only about half as much time (percentage-wise) is spent in Step 1 on the CM-5 than on the S81. Without the contention of processors for a single shared bus, the network of the CM-5 is able to move large amounts of data between processors more efficiently than the S81. Even though the pattern used in the first step may not be predictable beforehand, once the data begins moving from one processor to another the pattern of messages will not change until all records have been transferred. Distributed parallel systems that could quickly find optimal or near-optimal permutation routing schemes would be able to move the data blocks between processors with high efficiency.

4.5.2 Consumer/Producer Delays

Comparison of time spent in Step 1 and Step 3 on the CM-5 points to another drawback of message-passing systems. Step 3 involves the passing of very short messages for the binary search followed by several BLOCKSIZE messages for the linear search. The total amount of data sent across the network is no more (and should be less) than that sent in the first step, yet Step 3 takes longer to execute than Step 1.

The major reason for this phenomena is that records being sent in Step 1 are all being sent in parallel while Step 3 messages are being passed in a more haphazard way. Step 3 messages and blocks of records are being sent out by the second series processors only after

receiving a request from first series processors. This results in a marked waiting time between a processor sending a request for data and the data returning to the processor. While there is a time delay between sending a block of data and receiving its replacement within Step 1, such delays are going to be in direct relation to the communication characteristics of the hardware. The delays between sending and receiving messages in Step 3 will depend more on the number of consumer (first series) processors for each producer (second series) processor. The more consumers that need data from a producer, the longer each must wait between sending requests and receiving data.

4.5.3 Block Size Variations

One way to illustrate the impact of differing block sizes for messages on the CM-5 is to examine the amount of processing time taken for communication (Table 4.5) and non-communication (Table 4.6) by each step. As in Table 4.4, the times given in these two tables are the arithmetic mean of the maximums reported by a single processor over five randomly generated test runs.

Table 4.5 gives the time spent performing communication broken down by steps of the merge algorithm for block sizes of 10 and 100 keys. This includes the communication function call overheads as well as the waiting time needed before messages were sent or received. Step 1 is further divided into communication time specifically used in the bitonic sort of tail keys (bitonic) and the data movement due to block sorting (block). Step 3 communication is broken down for both the binary search for the sorted position of the tail key within the second series (binary) and the subsequent linear search to compute the displacement table

Table 4.5: COMMUNICATION TIMES BY STEP

Total number of keys	Block Size	Step 1		Step 2	Step 3		Step 4	Step 5
		bitonic	block		binary	linear		
524288	10	0.01	0.67	0.01	0.04	0.45	0.40	0.00
	100	0.01	0.08	0.01	0.04	0.07	0.10	0.00
2097152	10	0.01	2.68	0.01	0.04	1.77	1.58	0.00
	100	0.01	0.35	0.01	0.04	0.27	0.37	0.00
8388608	10	0.01	10.83	0.01	0.05	7.07	6.35	0.00
	100	0.01	1.47	0.01	0.05	1.07	1.49	0.00
33554432	10	0.01	42.96	0.01	0.06	28.87	25.39	0.00
	100	0.01	5.70	0.01	0.05	4.45	5.96	0.00

Table 4.6: NON-COMMUNICATION TIMES BY STEP

Total number of keys	Block Size	Step 1	Step 2	Step 3	Step 4	Step 5
524288	10	0.011	0.0002	0.059	0.074	0.105
	100	0.002	0.0002	0.054	0.066	0.105
2097152	10	0.042	0.0002	0.179	0.307	0.427
	100	0.007	0.0002	0.145	0.279	0.427
8388608	10	0.165	0.0002	0.640	1.234	1.726
	100	0.023	0.0002	0.510	1.118	1.726
33554432	10	0.667	0.0002	2.500	4.913	6.979
	100	0.090	0.0002	1.931	4.469	6.979

entry (linear). Step 5 needs no communication time since this is the final merge step and all computation is done locally. The times for Step 2 and the bitonic merge in Step 1 rely solely on the number of processors and not the size of the data. The binary search portion of Step 3 increases in proportion to the logarithm of the size of the data. The communication time for all other steps can be seen to change dramatically between the different block sizes. Consider the block data movement portion of Step 1. While it may take more time to send a message of 100 records versus 10 records, the difference is relatively small. The biggest savings exhibited for this step is that there are about one-tenth fewer messages to be sent and consequently one-tenth the number of messages to be received and synchronized.

Table 4.6 details the amount of time not spent in communication for each step of the merge code. The difference in block sizes has a direct relation to execution time. Again, in reference to Step 1, with fewer message to be sent and received, the number of iterations (and the attendant overhead) through the loop controlling the exchange of data is reduced to about one-tenth. The run time for Step 1 does not see a 90% reduction since the amount of time needed to move all records to and from a message buffer is included in this portion of the step times.

4.6 Some Lessons Learned

Distributed memory machines are the most prevalent parallel computer systems available today. With each different processor interconnection network, implementors of numerical codes can be faced with a different set of efficiency issues. The transformation of an algorithm

from the Sequent Symmetry S81 to the Thinking Machines CM-5 has given us the insight that few non-numeric codes will address these issues. In particular:

- Execution patterns of non-numeric codes do not map well to network topologies. Despite this, we have seen that large-scale data movement across such data networks can be more efficient than across a shared bus. However, because of the nature of message passing protocols, there can be a large amount of processing time lost waiting for message to be sent or received.
- Distributed memory machines, without regard to a specific network topology, offer a different variety of algorithmic solutions that are not well-suited for shared memory machines. (Shared memory machines may execute algorithms designed for distributed memory by “passing” messages within a portion of memory designated for this purpose. However, there are usually much more efficient methods to share data.) For example, implementations of a multiple processor binary search within the same data segment and the sharing of data via a consumer/producer relationship. These solutions often come with an additional cost. While message passing codes may be able to use a simpler algorithm than a shared memory counterpart, overhead and latency of communications used to implement the codes tend to increase execution times.
- Some thought must be put forth to reduce coding complexity and message passing. While our simple technique for identifying where communications must be made for shared to distributed conversion, this method can prescribe message passing scenarios that are overly complicated and/or impossible to implement. Knowledge of the al-

gorithm and data requirements will go a long way toward simplifying message passing coding and the number of messages needed to be sent and received.

4.6.1 Other Machines

Once one distributed memory code has been developed, translation to other message passing systems is relatively simple. A conversion of this nature needs only translate the code into the appropriate programming language version and recode all message passing functions to the target machine. This will generally amount to nothing more than a change in the function name and a reordering of the parameters. In the case where functions used on one machine are not supported on another (e.g., parallel scan and reduction), these will need to be explicitly coded by the programmer.

With the standardization of the Message Passing Interface (MPI) [27, 52] and the widespread acceptance of this standard by parallel system manufacturers and vendors, once a program has been converted to MPI, it will be able to run on many different platforms. However, experience suggests that modifications for efficiencies relating to the underlying hardware may still be necessary.

Chapter 5

Parallel Benchmarks and Comparison-Based Computing

5.1 Introduction

Only in recent years have non-numeric algorithms been considered for inclusion in parallel and supercomputer benchmarks [6, 18, 55]. Past benchmarking suites have been primarily concerned with a parallel system's aggregate speed of performing floating-point operations. As the field of parallel computation matures, however, the range of codes executed on parallel machines will likely include a wide assortment of non-numeric routines.

Counting sort and radix sort have been proposed as non-numeric benchmark candidates [6, 55]. Unfortunately, these simple algorithms are not very representative of operations carried out in wide assortments of non-numeric applications. Both of these sorts exhibit completely predictable data access patterns and, even worse, can require up to an exponential amount of extra memory. Thus, in order to predict more accurately the performance of non-numeric algorithms, a comparison-based, in-place benchmark may be more telling.

In this chapter, we examine the effectiveness of the G-L merge (and hence sort-by-merging) approach. We first survey several known benchmarking efforts, including those for sequential and parallel computational models. We next focus on the needs of a suitable non-numeric parallel benchmark. Previously-proposed benchmark alternatives are considered in this light and the relative merits of in-place merging are discussed. We report the results of large collections of experiments conducted on a wide range of parallel machines, including both SIMD versus MIMD and shared versus distributed memory designs. Besides a direct comparison between the different machines used, we compare our results to those of numeric benchmark suites. We close with a few observations and related remarks.

5.2 Previous Efforts

5.2.1 Sequential Benchmarks

We briefly examine some of the better-known sequential non-numeric benchmarking algorithms that are available. We know of no parallel implementations or performance results for any of the codes described within this section.

Dhrystone

The Dhrystone benchmark [62] is a synthetic collection of operations based upon a literature survey of the distribution of language features most used in non-numeric, system-type programming. The major computational concentration is on string functions. Originally implemented in Ada, more recent versions have been coded and distributed in C [63]. The

Dhrystone benchmark is intended to measure integer performance on small machines with simple architectures.

Stanford Small Programs Benchmark Set

Weicker [64] makes mention of the Stanford Small Programs Benchmark Set. Used for the first comparisons between RISC and CISC processors, this benchmark is a collection of small programs brought together at Stanford University by John Hennessy and Peter Nye. In addition to two floating-point routines, there are eight integer codes including quicksort, bubble sort and tree sort. These units have been collected into a single C program and made available through informal channels.

Aburto's Collection

Alfred Aburto at the Naval Ocean Systems Center, San Diego, maintains, collects and publishes results for a number of separate non-numeric codes that continue to be used for benchmarking systems. These include Heapsort, Hanoi (solves the Towers of Hanoi puzzle), NSieve (finds prime numbers) and Sim (locates similarities between DNA sequences). Also available is Fhourstone, a program to solve positions from the game Connect-4 that tests hashing and random access performance, recursive alpha-beta searching and other scalar operations involving array and table computations. These benchmarking results and routines are available through several on-line sources.

EDN Benchmarks

This collection of programs was developed at Carnegie Mellon University and published by EDN in 1981 [25]. The intent of the benchmarks was to measure the computational speed of microprocessors without measuring the quality of the compiler. For that reason, the original codes were written directly in assembly languages of various microprocessors, though a subset of the original benchmarks are now available in C. This subset includes routines for string search, linked list insertion and quicksort. There is no formal mechanism for distribution.

5.2.2 Parallel Benchmarks

In testing their Threaded Abstract Parallel Machine model of computation on the Thinking Machines CM-5 and the MIT J-machine, Spertus et al. [53] make use of a simple quicksort. This algorithm, along with others used for comparison of the two machines, were written in the parallel language Id90. Francis and Mathieson [22] put forth a parallel sort algorithm for benchmarking shared memory machines. There are more individual benchmarking examples used to rate the performance of specific hardware systems or architectures in the literature. Bhuyan and Zhang bring several of these examples together in [10].

Attempts to develop and make available a standardized parallel benchmarking suite are in their infancy. These attempts include the EuroBEN Group [60], the GENESIS Project [1], PERFECT Club Benchmarks [9], NAS Parallel Benchmarks [6] and the vendor-supported SPEC Benchmarks [18]. Recently, a number of researchers interested in benchmarking met and formed the PARKBENCH Committee [32]. The goal of this group is to make available a

hierarchy of benchmarking codes (low-level, kernels and compact applications) that test the capabilities of scalable distributed memory computers while maintaining strict guidelines on an acceptable benchmarking methodology to ensure that performance evaluations are meaningful across a variety of machines. Some of the routines within the current PARKBENCH collection come from the NAS Parallel Benchmark and the GENESIS Benchmark suites.

All of these benchmarking programs and suites are geared toward scientific computations. This, in turn, translates to codes devoted to computations involving single and multiple precision floating-point operations. There are, however, some non-numeric exceptions included within two of the suites.

The NAS Parallel Benchmarks [6] is a collection of different algorithms found in actual computational fluid dynamics calculations. The entire suite includes five parallel kernels and three complete applications. One kernel is a sort, needed as part of certain particle codes, whose data set is made up of eight million nineteen-bit numbers, each generated by taking the average of four random numbers ranging between 0 and 2^{19} . This benchmark is intended to measure integer computation and communication performance.

In [55], Thearling and Smith note that the NAS sorting benchmark does not go far enough with either the size of the data file to be sorted or the characterization of the data set used. They observe that the sorting of a billion or more keys has been achieved on at least two parallel systems. Thus, the data set size of the NAS benchmarks is already dwarfed by the memory capacity of certain parallel machines. They propose instead a radix sort, and present performance results using a CM-5 with varying numbers of processors and several

data distributions.

Within the integer computation suite, CINT92, of the SPEC Benchmarks [18] are routines for generating and optimizing Programmable Logic Arrays, solving a nine queens' problem with a Lisp interpreter and compressing input files with the Lempel-Ziv encoding. Another code, *eqntott* [34], that translates logical Boolean expressions into truth table equivalents, makes use of a sorting procedure.

5.2.3 Database Benchmarking

In contrast to benchmarks targeted at specific types of computations or performance of particular hardware subsystems (e.g., I/O latency, cache access, page swapping), benchmarks for database management systems use higher level measures such as tuple retrieval time or the number of queries satisfiable within a given amount of time. Integer or character computations are used for evaluation and often involve comparisons of quantities or movement of data. Such computations may be repeated in the processing of a single transaction or query.

Two of the better-known database benchmarks are the Wisconsin benchmark [16] and the TPC benchmark [50]. The former uses a synthetic database with a set of queries designed to measure performance of decision support systems; the latter measures transaction processing using a large number of small transactions. Both benchmarks were originally designed for serial systems, but are adaptable for use on parallel ones. The performance evaluation of the Gamma parallel database machine under the Wisconsin benchmark is included in [16]. TPC comes in three versions: one for online transaction processing with a LAN or WAN, one for online transaction processing with no network and one for online business transaction

processing (which includes batch transactions, data entry errors that cause some transactions to abort and several other features not found in the other two suites).

Citing the need for a finer level of granularity than is generally available from serial benchmarks, McCann and Bell [46] have developed a hybrid benchmarking model. There are two stages to the model: serial and parallel. The serial stage measures resource consumption by modeling the system in terms of input, output, comparison and other low level operations. Queries in this model are analyzed in terms of CPU time used to complete a set of elementary operations including fetching a page from disk to memory, comparing either a single integer or character and outputting a tuple that satisfies conditions set with a given query. The parallel stage models database operations as a network of communicating sequential processes. This model uses CPU times for serial elementary operations and is able to represent the traffic congestion of real parallel systems under a real workload. McCann and Bell present sample simulation results run on a two transputer system. Implementation of the model was made possible through the high-level queuing network package Network II.5 (CACI Products Company).

5.2.4 In Summary

None of the non-numeric exceptions within scientific benchmarks adequately deals with the measurement of non-numeric computations. Database benchmarks are a little better, but are too specialized for use in benchmarking generic non-numeric performance on parallel machines. Thus, we shall employ the in-place G-L merge as our algorithm of choice. An explanation of this selection follows.

5.3 Defining a Suitable Benchmark Algorithm

Gray [26] lists four criteria for domain-specific benchmarks. Though this list was aimed at database systems, we argue that it can easily be extended to more general non-numeric parallel environments. We list Gray’s criteria below, and describe how each can be applied to parallel benchmarks in general, and to non-numeric processing in particular.

Relevance. The purpose of benchmarking is to measure the performance of machines over computations that are typical of the work expected to be routinely performed. For parallel machines, this applies not only to the processing of data, but also to the patterns of data movement that are anticipated in the execution of production applications.

Portability. Parallel benchmark algorithms should not rely on the particular features of any single machine. Implementations should be independent of the number of processors, network topology and available programming language features (up to those machine-dependent functions necessary for proper execution). Thus a benchmark should provide a “level playing field” for comparing different machines. Because a wide range of execution modes are available across different architectures, benchmarking algorithms should also be implementable on both shared versus distributed memory and SIMD versus MIMD systems. When implementing an algorithm across architectural paradigms, care must be taken to keep computations as equivalent as possible within given machine constraints.

Scalability. An algorithm should be scalable to differing numbers of processors and memory sizes. As computer architectures continue to evolve, the longevity and usefulness of benchmarking codes will lie in large part in their adaptability to new and larger systems as they become available.

Simplicity. Benchmarking routines must be understandable, else they lack credibility and are not readily accepted by large numbers of potential users. Portability and scalability may be hindered if an implementation is too complicated. The goal is to select algorithms that are complex enough to tickle the desired range of architectural features. However, they must not be so intricate that the code is indecipherable.

5.3.1 Non-Numeric Operations

We have identified the following operations as being desirable within a non-numeric benchmark:

- comparison of data within local memory,
- movement of data within local memory,
- comparison of data across processors,
- movement of data across processors, and
- global scan/reduction operations.

Parallel merge or sort algorithms probably best fit this bill. It is important, however, that the movement of data across processors be rather unpredictable and driven by the data values themselves. This is in marked contrast with numeric benchmarks, which generally have inter-processor data movement schemes built into the algorithms. Unpredictable data movement

is typical of non-numeric applications, and is much more apt to create network congestion in parallel systems. Determining just how effectively such movement is automatically handled is another benefit derivable from non-numeric benchmarks.

There are several benchmarking kernels and proposed kernels that do sorting or merging. We look at these algorithms in turn, plus Valiant's parallel merge, with an eye toward how well each measures up to our expectations. In all cases, we consider implementations of the selected algorithms that are designed for a fixed, finite number of processors. Algorithms designed with only the PRAM model in mind often ignore this real-life restriction, and describe idealized computations done with a number of processors related to the size of the input data set. Implementation of such algorithms can often be accomplished directly by using each physical processor to emulate a number of virtual processors required by the PRAM algorithm. Unfortunately, as we have seen, this simple-minded approach does not scale with limited memory.

5.3.2 The NAS Parallel Sort

The NAS Parallel Integer Sort benchmark is based on the serial code of the benchmark kernel and is available in HPF and Fortran 90 as well as specialized Intel iPSC/860 and Thinking Machines CM-2 versions. For the latter two, both C and FORTRAN versions are available.

This benchmark measures the time needed to rank randomly generated keys and perform a partial check for correctness. The two step process of ranking and checking is repeated several times. After the timing is done, the data is moved to its sorted position and completely verified for accuracy. This data movement is not part of the benchmark timing.

While such a counting sort algorithm is a part of larger scientific codes, and thus a valuable kernel within the NAS benchmarking suite, its suitability as a non-numeric performance measure is very much in doubt. There is no direct comparison of keys (ranking is accomplished by counting), data movement is not even timed, and no global scan/reduction operations are involved.

5.3.3 Radix Sort

Parallel radix sort [41] implemented on a finite number of processors first evenly divides the records to be sorted among the processors. Each processor ranks the given keys based on successive non-overlapping segments of bits within each key starting at the least significant portion of the key and working up to the most significant. For each segment, this ranking is done by generating all the possible bit patterns of the specified size in numerical order. For each pattern, the number of keys that match the pattern within the target segment are counted and given a rank index based on the key's position in the list, the number of previously ranked keys and the number of keys whose segment matches the current pattern. Parallel scan operations are used to compute the ranks of keys across processors. Once the rank of all keys is known, the records are permuted based on this rank. The ranking and data movement is repeated for the next bit segment until the entire key has been processed in this way.

In principle, radix sort is perfectly data balanced. That is, the amount of records and workspace initially allocated to each processor is unchanged throughout the execution of the algorithm. Under a shared memory paradigm, the movement of data is easily done by

moving records into temporary array elements indexed by each key's rank. After all records are moved, the algorithm can either move data to the corresponding elements of the original array before proceeding to the next bit segment or merely swap roles between the temporary holding array and the original data array for each bit segment used. The former method would require both global and local data movement while the latter only global movement.

Unless programmed carefully, movement of records between distributed memory processors can end up in deadlock. Even with a careful implementation the data movement can degenerate into a "token ring" wherein only a single record is in transit between processors. Upon receipt and local storage of the record, the receiving processor is then able to send out a single record to another processor, and so on, until all records have been moved. This adverse scenario depends upon how message passing routines are implemented, key values and their distribution among the processors.

A much more fundamental problem with radix sort is that it can only deal with keys whose bit patterns directly determine their lexicographic ordering. Records whose keys contain non-standard characters or formats cannot be easily handled. Records whose keys are determined from a combination of two or more data fields may not be handled at all. Furthermore, global data movement is really nothing more than the permutation of data through a network. More effective algorithms [20, 36, 47, 54] are available if one wishes to measure this very limited type of data movement.

5.3.4 Valiant's Parallel Merge

Converting the algorithm of Valiant [58] to a computational model with a fixed number, p , of processors is fairly simple. Instead of being able to distribute $\lfloor \sqrt{nm} \rfloor$ processors to merge $\sqrt{n}$ smaller sublists, we merge $O(p)$ sublists on the p processors by dividing the first list into p sublists and locating the points within the second list where the last element of each sublist would be placed if these elements alone were merged into the second list. These points are the endpoints of the sublists within the second list that are merged with the original sublists from the first list.

This parallel merge performs comparisons across sublists, local key comparisons for the final merges and movement of data on distributed machines to locate elements from the second list within the proper processor memory for the final merge. Shared memory implementations need only update pointers to locate the sublists from second list blocks that are to be merged with the first list sublists. The local merge moves data between the input array and an additional auxiliary array.

In general, this merge is not well load or data balanced. One can easily construct data sets that will restrict the entire final local merge to be executed on a single processor. On distributed memory machines with the two original lists spread throughout the local memory of the processors, if the data capacity of each processor is initially at or near capacity, the data redistribution of the second list sublists may not have the required space available on some of the processors. Data sets can, of course, be contrived to ensure that a machine is load balanced and that the data capacity of distributed processors will not be exceeded. In

this case, however, memory access and communication patterns are predictable and unlikely to be representative of arbitrary computations.

5.3.5 In-Place Merging

The algorithm of [28] covers the five desired operations very well, and is easily scalable to any number of processors or memory size. During the course of execution, keys are compared and records moved between processors, global scan and reduce operations broadcast the positions of certain records, and data is compared and moved within a processor's memory. Two very different kinds of data movement are carried out between processors. The first involves moving an entire block of data initially assigned to a processor into another block (Step 1). Since such a large amount of data is being transported to a single destination and all processors are participating, this operation can be done by moving large portions of the block in a single message with distributed memory or synchronous copying with shared memory. The second type of global movement shifts records within small subsets of processors (Step 4) in preparation for a local merge. The number of records between subsets and between processors within the same subset varies greatly, making for unpredictable, data-driven communication patterns between processors and differing message sizes on distributed memory machines.

5.4 Implementation and Timing Results

Table 5.1 contains hardware, operating systems and compiler details on each of the machines we have used within our study. See § 2.4 for more details on each machine.

Table 5.1: MACHINES USED IN STUDY

<u>Connection Machine CM-5 (SIMD)</u> Operating System: SunOS Release 4.1.2; CMOST Version 7.2 Compiler: C* Driver Version 7.1 Final Rev f2000 Processors: SunSPARC1 (plus 4 vector units per node) Memory per Node: 32 Mbytes	<u>MasPar MP-2</u> Operating System: ULTRIX V4.3 (Rev. MP-3.22) Compiler: MPL Version 3.2.14 Processors: proprietary RISC Memory per Node: 64 Kbytes
<u>Connection Machine CM-5 (MIMD)</u> Operating System: SunOS Release 4.1.2; CMOST Version 7.2 Compiler: SunOS C Compiler with CMMD Extensions Processors: SunSPARC1 Memory per Node: 32 Mbytes	<u>Intel iPSC/860</u> Operating System: iPSC/860 UNIX Sys V 3.2, NX 3.3.2 Compiler: icc/NX Sun4 Rel 4.0 Processors: Intel i860 Memory per Node: 8 Mbytes
<u>IBM SP2</u> Operating System: AIX 6000 Compiler: AIX C Compiler with MPL Extensions Processors: RS/6000 (POWER2 architecture) Memory per Node: 128 Mbytes	<u>Intel Paragon</u> Operating System: Paragon OSF/1 Release 1.0.4 Server 1.3 R1.3 Compiler: icc/Paragon Version R5.0.1 Processors: Intel i860 XP Memory per Node: 16 Mbytes

5.4.1 Methodology

All programs were written in C to promote portability. Accordingly, no low-level or machine-specific optimizations were used. In general, no specialized communication routines other than send, receive and broadcast were employed. On distributed memory architectures, only global synchronization, send, receive, one-to-all broadcast and a limited number of informational functions were allowed since all target machines support these operations. Thus, no reliance was placed on any one architecture, memory hierarchy, or connection topology.

Records (and consequently keys) are thirty-two-bit integers. Two sorted lists are created by dividing the data and processor sets in half and assigning an equal portion of the data to each processor. Each data set is divided into several (typically fifty) segments and the data generated within each segment has a different, random density of duplicate key values. By varying the density of duplicates we are attempting to model real-world data sets that would likely be merged or sorted and whose key values aren't all equally probable. Compilations were done with maximum optimization available through compiler flags.

5.4.2 Machine Comparisons

Numeric benchmarks measure the number of floating-point operations executed per second. For non-numeric computations such a measure is meaningless. Also, exact counts for numbers of statements executed are not easily calculated *a priori* and rely entirely on the initial distribution of data. Thus, a measure on the number of machine instructions executed per time unit (e.g., millions of instructions per second (MIPS)) would be nearly impossible to

compute for such codes. However, based on the performance metrics outlined in [31], a more useful measure for non-numeric algorithms such as ours is the number of records handled per time unit.

Thus we use the average number of records merged per second (rec/s). This value is computed by dividing the total number of records by the total time taken by the merge. Unlike flop/s, rec/s is an amortized metric in that after one second of execution of the merge it is not the case that the given number of records will be in their final merged position. However, such a metric directly relates the number of records within the data set to the execution time of the code. Not only does this give a standard measure by which we can compare different machines with different numbers of processors, but we can also compare different amounts of data on the same machine to determine how variations in load affect performance.

Table 5.2 lists the average Mrec/s (millions of records per second) performance we observed for each machine over a range of data set sizes. The data set sizes used in our tests are the exact powers of two contained within the ranges shown plus those sizes midway between successive powers of two. The minimum data size was chosen as that size which began to show significant differences from smaller-sized data sets (and thus large enough to overshadow time required for machine-dependent overhead); the maximum was the data size that was a power of 2 or midway value and would still fit within the memory of the target machine (or, in the case of the CM-5 under SIMD execution, this was the data size that would complete 5 test runs in less than a day). Timings on each machine are taken from the average of five different, random test runs for each file size.

Table 5.2: AVERAGE PERFORMANCE OF EACH MACHINE

Machine	Number of Processors	Range of File Sizes Merged	Average Mrec/s
IBM SP2	16	32768 – 268435456	4.621
Intel Paragon	32	131072 – 33554432	3.650
IBM SP2	8	32768 – 134217728	2.174
Intel Paragon	16	65536 – 16777216	1.984
Thinking Machines CM-5 (MIMD)	32	32768 – 50331648	1.664
Intel iPSC/860	32	131072 – 50331648	1.515
Intel Paragon	8	32768 – 12582912	1.092
Intel iPSC/860	16	32768 – 25165824	0.832
Intel iPSC/860	8	32768 – 12582912	0.489
MasPar MP-2	4096	32768 – 50331648	0.474
Thinking Machines CM-5 (SIMD)	32 (128 VU)	32768 – 3145728	0.002

The graph in Figure 5.1 depicts the entire set of averaged run times we observed over the course of our experiments. Note that the x axis uses a logarithmic scale. Also, since the relative values of the SIMD CM-5 are so low, they are indistinguishable from the x axis.

The graph in Figure 5.2 shows the range of values that make up a selected portion of the average data points used in Figure 5.1. We feel that this range supports our use of averages in the presentation of our results for these experiments rather than reporting a single data set execution time. Looking closer we discover that, even though the same 5 data sets were run on each machine for each data set size, the percentage differences between the average and the low and high execution times can vary widely between machines. For example, merging 2097152 records on the SP2 the low and high execution time range from -20% to +40% of the average while on the MP-2 the differences range from -5% to +2%. While a single data set for any given data set size used for benchmarking would be the best situation, it will

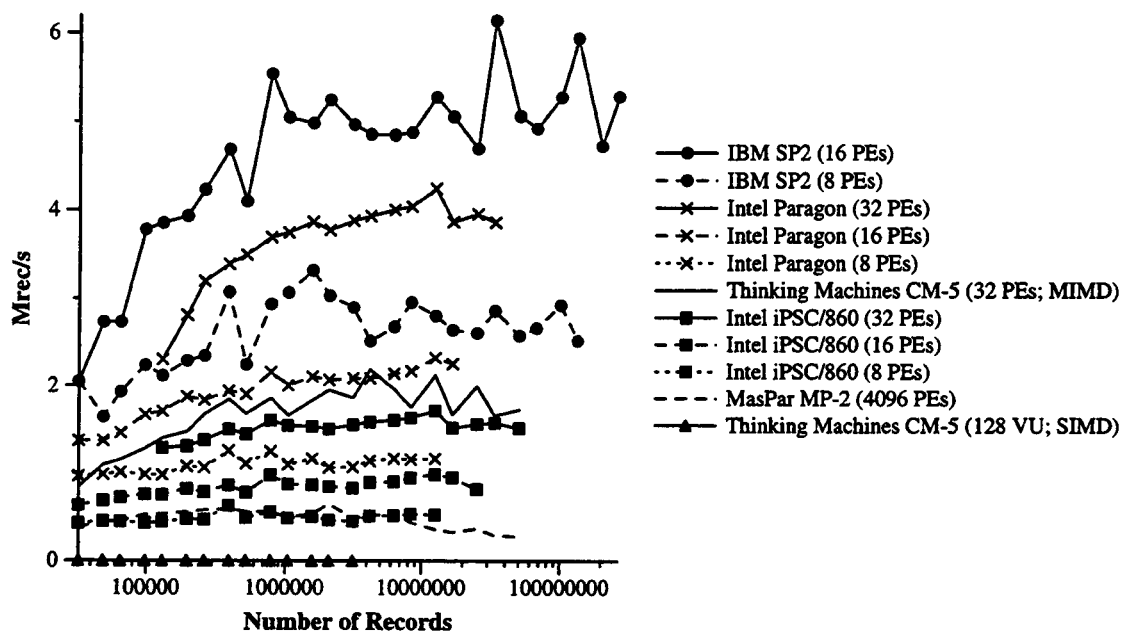


Figure 5.1: DETAILED PERFORMANCE OF EACH MACHINE

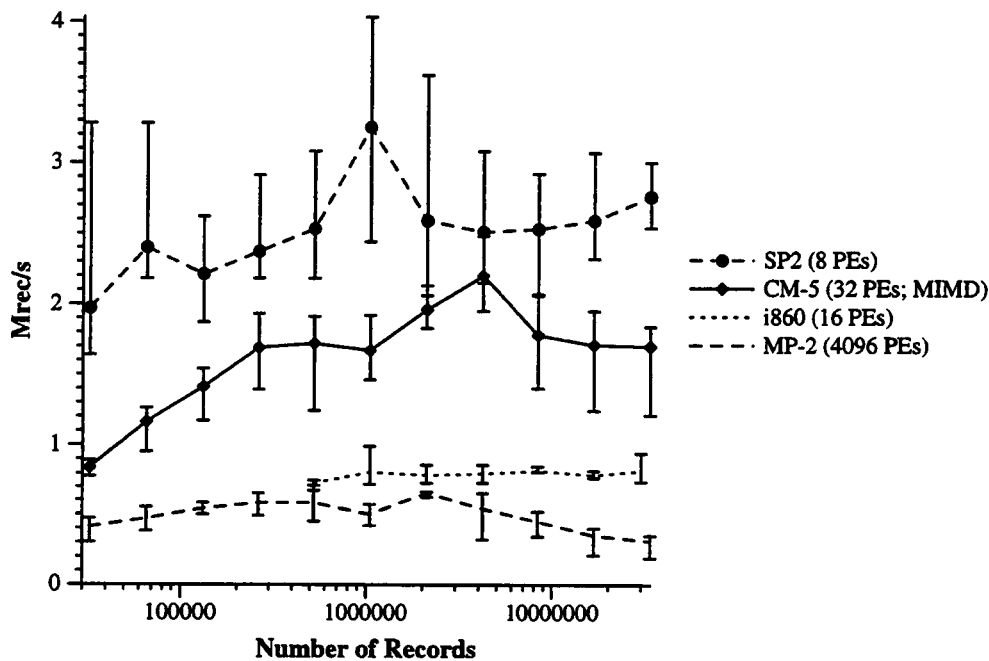


Figure 5.2: RANGE OF EXECUTION TIMES FOR SELECTED MACHINES

require a more extensive investigation to determine such data sets that will be fair across many different execution platforms.

5.4.3 Memory Capacity

Memory capacity is another critical measure. We have observed before [12] that memory management schemes can exhibit unexpected behavior at or near capacity. Table 5.3 lists maximum data memory capacity and corresponding machine performance for three sample configurations: MasPar MP-2 with 4096 processors, Thinking Machines CM-5 (MIMD) with 32 processors and Intel iPSC/860 with 32 processors. The percentage listed in the rightmost column are the ratio of maximum data set size allowed versus total memory available.

Table 5.3: MEMORY CAPACITY MEASURE AND PERFORMANCE

Machine	Maximum file size	Mrec/s	Memory Utilization
MP-2	65404928	0.236	97.5%
CM-5 (MIMD)	243760960	0.312	93.0%
iPSC/860	59959808	1.428	91.5%

5.4.4 Benchmark Comparisons

We note a vast difference between the relative rankings of machines in our study and their rankings under the LINPACK Benchmarks. Using the notation of [19], we let R_{max} denote billions of floating-point operations per second (Gflop/s) measured for the largest problem run on each machine. For comparison, we use Mrec/s values. The selected results from both

benchmarks are shown in Table 5.4¹; numbers in brackets represent relative rankings. Among other things, Tables 5.2 and 5.4 both illustrate that SIMD architectures can be extremely unfriendly to non-numeric algorithms.

Table 5.4: COMPARISON OF LINPACK AND G-L MERGE BENCHMARKS

Machine	Number of processors	LINPACK R_{max} (Gflop/s)	G-L Merge (Mrec/s)
Intel iPSC/860	32	.64 [2]	1.428 [1]
MasPar MP-2	4096	.374 [3]	0.236 [2]
Thinking Machines CM-5 (SIMD)	32 (128 VU)	1.9 [1]	0.001 [3]

We also note differences between our results and those of the NAS Parallel Benchmark Integer Sort kernel. We have taken from [5] the raw execution times for 2^{23} keys and converted them to Mrec/s. See Table 5.5.

Table 5.5: COMPARISON OF NAS INTEGER SORT AND G-L MERGE BENCHMARKS

Machine	Number of processors	NAS IS (Mrec/s)	G-L Merge (Mrec/s)
IBM SP2	16	3.077 [1]	4.877 [1]
IBM SP2	8	1.678 [2]	2.956 [3]
Intel iPSC/860	32	0.326 [4]	1.630 [4]
Intel Paragon	32	1.074 [3]	4.039 [2]

In addition to the differences between our rankings and those of LINPACK and NAS, we observe a quantitative difference between relative machine performances. As an example, consider that the Intel iPSC/860 outperforms the MasPar MP-2 by a factor of less than two under LINPACK, but by a factor of over six in our study.

¹The selection of these particular machines was based solely on the fact that each had an entry in the LINPACK report for the same numbers of processors on which our in-place merge code was run.

5.5 Discussion

By implementing a single algorithm across a variety of parallel platforms, we have investigated the behavior of these machines in supporting representative non-numeric codes. We shall not provide detailed explanations for each of the specific numbers we have obtained; a variety of intricate architectural features can come into play. We have in fact aimed instead to eschew optimizations that favor any one particular cache coherence scheme, network topology, memory hierarchy or other possible machine-specific performance factor. We recognize that such factors are important, and that they can directly affect how well a given system will perform. We are much more concerned, however, with issues of portability, scalability and fairness.

Even with the results we have depicted, we would not propose a wholesale replacement of other merge and sort kernels in parallel and supercomputing benchmark suites. Rather, we would argue for the addition of representative comparison-based algorithms, such as the one we have employed, to cover a much wider range of applications.

Chapter 6

Algorithmic Improvements

6.1 The Problem

On SIMD machines, the G-L merge must execute two separate local merges in those blocks that hold a breaker. Due to lockstep execution, these merges are done in sequence. On distributed memory machines, a large portion of execution time during the first four steps is spent on communication. Data from Tables 4.5 and 4.6 show us that over 70% of execution time is related to communication. Step 1 spends 98% of its time in block data movement. The communication involved in the linear search in Step 3 accounts for 42% of execution time on small data set sizes, increasing to almost 70% for large data sets.

In this chapter we examine modifications to portions of the G-L merge algorithm tailored to the characteristics of different execution paradigms. The goal of these changes is to reduce execution time on particular architectures. Specifically, we will address methods to rearrange data blocks in order to eliminate the need for a second local merge phase on SIMD machines, reduce the amount of data communicated between processors and reduce the number of times each record is communicated on distributed memory architectures. As with our previous

implementations, no machine-specific characteristics or functionality are considered. All of the following modifications are general enough to be applied to any machine with the execution paradigm under consideration.

6.2 SIMD Algorithmic Improvements

A major detriment of non-numeric SIMD processing can be the lockstep execution. As we have seen, this leads to the execution of the two different sequential in-place merge algorithms one after the other in the final step of the G-L merge. Because of the variety of sub-block sizes inherent in the data, and because processors need only execute one or the other merge, at any time during this step there is a high probability that processors are idle. In addition, processors holding a breaker may have to again execute one of the two merges on those records to the right of the breaker. This necessitates the execution of two phases of two merges (four merges in total) in sequence to complete the final local merge step.

Below we give two methods to perform the local merge step of the G-L merge on all processors in one merge phase (one of two merges on each processor). Each repartitions the data for the final merge step of the G-L merge such that processors holding a breaker will not need to execute an extra merge phase. This reduces the total number of local merges from four to two.

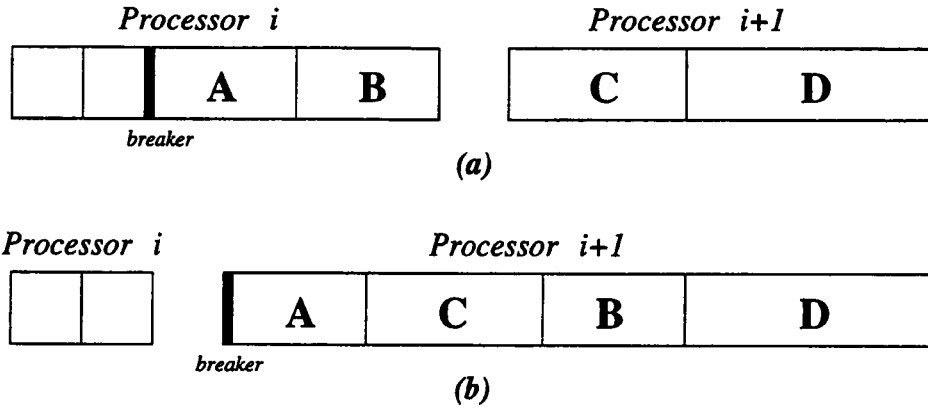


Figure 6.1: FIRST METHOD ILLUSTRATION

6.2.1 First Method

If no first series records, or alternately, all first series records, within a second series block (i.e., records to the right of the breaker) were moved during the record movement step (Step 4), no local merge is required on these records. However, if a fraction of the records were moved from the first series block of a second series processor, the resulting two lists must be merged.

Our first method will reallocate processor block boundaries. This will enable the sublists to the right of the breaker in second series processor P_i to be merged at the same time as the sublists of P_{i+1} . If P_i contains a breaker, label the two sorted sublists located to the right of the breaker as A and B. P_{i+1} contains two sorted sublists, which we label C and D. (See Figure 6.1 (a).) If P_{i+1} is a first series processor, the right boundary for sublist D will be the tail; if P_{i+1} is a second series processor, the right boundary of sublist D will be its breaker.

Before the local merges are executed, P_{i+1} swaps the positions of sublists B and C. The

sublists are then arranged ACBD. (See Figure 6.1 (b).) While P_i merges the two lists to the left of its breaker, P_{i+1} merges sublist AC with sublist BD. This rotation of record blocks and boundary extension allows the merge of the records to the right of the breaker during the same merge phase as the records to the left of the breaker. Thus, the need for an extra execution of local merges is eliminated.

We establish the correctness of this algorithm by first noting that each of the sublists A, B, C and D are sorted. We also note that all keys within A are less than or equal to all keys in both C and D. Similarly, all keys within B are less than or equal to all keys in both C and D. From this we conclude that the two concatenated sublists, AC and BD, are both in sorted order and will be merged correctly with the appropriate serial in-place merge.

6.2.2 Second Method

Taking the idea of rotating blocks in order to reduce the number of local merges required on SIMD machines, we present a more general scheme. If P_i contains a breaker, label the two sorted sublists to the left of and including the breaker as E and F. The two sorted sublists located to the right of the breaker are labeled as G and H. (See Figure 6.2 (a).) Before the local merges are executed, P_i swaps the positions of sublists F and G. The sublists are then arranged EGFH. (See Figure 6.2 (b).)

Because of the original position of the breaker between the list pairs EF and GH, we note that all keys within E are less than all keys in G. Similarly, all keys within F are less than the keys in H. Thus, sublists EG and FH are both relatively sorted. P_i need only merge these two local lists with the appropriate serial in-place local merge to complete the final step

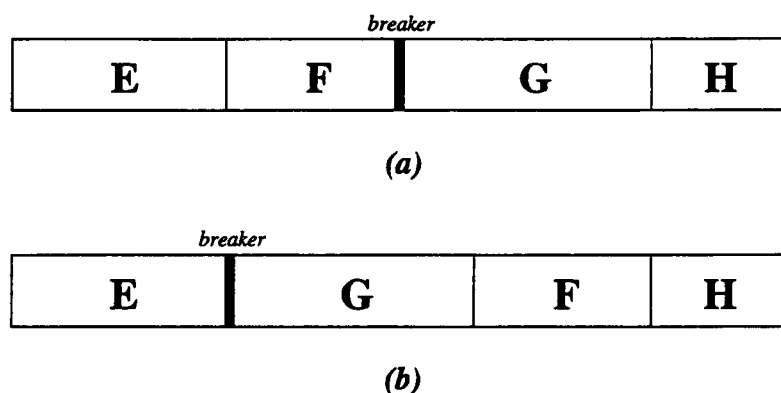


Figure 6.2: SECOND METHOD ILLUSTRATION

of the G-L merge. As above, this scheme of internal list rotation eliminates the need for a second set of local merges on SIMD machines.

6.2.3 Analysis

While our first method initially appears to have an advantage of reducing the amount of time spent in the local merges, this is not necessarily the case. The number of procedures and functions that must be executed has been reduced, yet the time taken to merge the newly formed lists may not be reduced dramatically. Because of lockstep execution, the time of each local merge will be the maximum time needed to merge any of the list pairs that are being merged. Using the reallocation of processor block boundaries scheme (§ 6.2.1), the worst-case number of records that will be merged will approach twice the size of the original data blocks. Thus, the local merge would execute two merges on data sizes almost twice the data block size ($2 \times [2 \times O(n/p)]$) instead of four merges on data sizes no greater than the original block size ($4 \times O(n/p)$).

Another drawback of this first method is that it will only work on shared memory SIMD machines. A distributed memory programming model (e.g., the MasPar MP-2) cannot efficiently support this modification since it requires access to data not local to a processor's memory.

Our second method (§ 6.2.2) keeps all records local to the processor that is handling the records. This version of the local merge step is implementable on both shared (Thinking Machines CM-2 and CM-5) and distributed memory (MP-2) models. In addition, since the total lengths of sublists to be merged are no greater than original block lengths, a reduction of execution time is readily apparent.

This second method could be adapted to run on both shared and distributed memory MIMD machines. However, since there is no execution conflict between MIMD processors for local computations, we feel that the amount of savings would be negligible.

6.2.4 Timing Results

Tables 6.1 and 6.2 give comparisons of the final merge step execution times between the original SIMD version (Original) and the above methods (First Method, Second Method). Table 6.1 shows average execution times generated on a 32 node CM-5 for all three algorithms. The first method actually adds time to the execution time of the last step. We feel that this is directly related to the difference in the architecture of the machine and the programming model. Since the CM-5 has a shared memory programming paradigm in SIMD mode, we were able to implement this method. However, the machine is designed with distributed memory; adjusting the processor block boundaries for this step involves access of data not

local to the processors. The second method fares much better on the CM-5. All memory references are local and only a single merge phase is required to complete the step.

Table 6.1: CM-5 SIMD IMPROVED LOCAL MERGE TIMING

Number of keys	Time (seconds)		
	Original	First Method	Second Method
32768	12.814	15.483	9.772
65536	30.359	35.268	24.207
131072	64.479	75.860	50.718
262144	138.627	161.842	111.418
524288	289.339	337.933	253.386
1048576	595.752	696.280	568.806
2097152	1289.765	1454.368	1091.916

The times in Table 6.2 were generated on a 4096-processor MP-2 and are the average of five randomly generated test runs for each data set size. Only the second method could easily be implemented on the MP-2 since the machine has a distributed memory architecture. As with the CM-5 implementation, this method yields lower execution times for the final merge step. Figure 6.3 displays this data graphically.

Table 6.2: MP-2 SIMD IMPROVED LOCAL MERGE TIMING

Number of keys	Time (seconds)		Percent of Improvement
	Original	Second Method	
524288	0.526	0.381	28.3
1048576	1.297	0.963	26.2
2097152	2.973	2.185	26.3
4194304	7.136	4.716	33.9
8388608	17.207	10.732	37.7
16777216	42.690	25.837	39.8
33554432	100.501	61.926	38.4

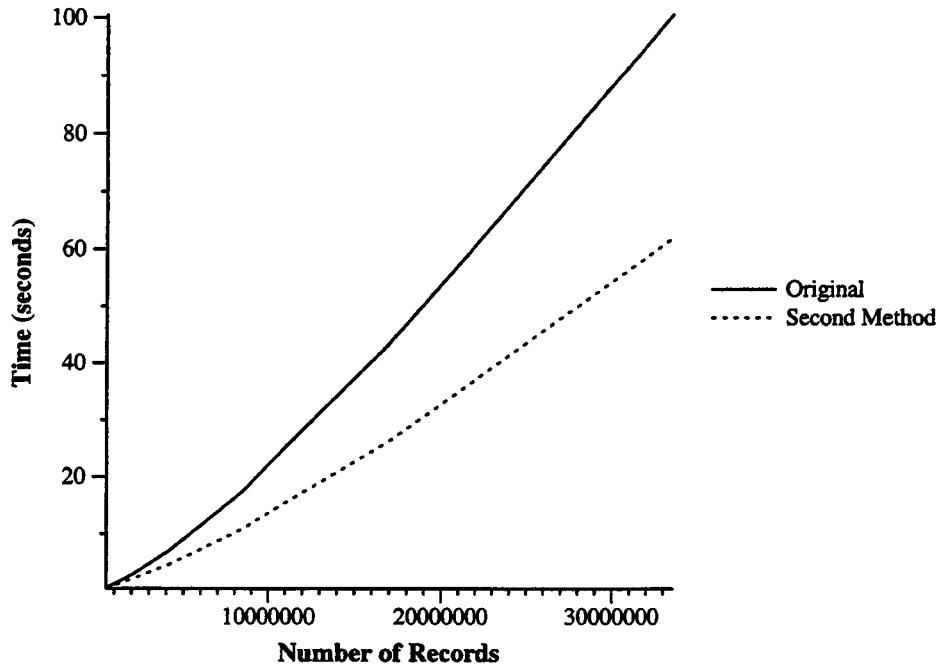


Figure 6.3: SIMD LOCAL MERGE TIME FOR IMPROVED ALGORITHM (ON MP-2)

6.3 Displacement Table Computation

The purpose of the displacement table entry for each processor is to determine the number of records from the first series block to be moved out and replaced in preparation for the local merge step. The difference between a processor's displacement table entry and its neighbor's entries indicate exactly how many records will be transferred to other first series blocks and how many records will be moved to the second series block. Computation of the table entry is performed by each individual processor without regard to the other first series blocks associated with the same second series block. Put another way, the number of records that must be swapped between the second series block and the first series block in P_i to ensure that all records in P_i precede those in the second series block is the displacement table entry,

E_i .

For the EREW PRAM version of the G-L merge, table entries are calculated with a two-part algorithm: a phased merge followed by a linear search. Since exclusive access is a by-product of distributed memory architectures, we were able to implement a simpler binary search in place of the phased merge. This left the time consuming linear search.

After we had pondered possible alternatives to the linear search it was realized another binary search could be used instead. In fact, without the explicit need to devise an algorithm that fit the restrictions of the shared memory model, the following approach suggested itself.

The goal of a search is to find a number, k , such that $A[hi-k]$ in P_i is greater than or equal to $A[k]$ in the associated second series block, and $A[hi-(k+1)]$ in P_i is less than $A[k+1]$ in the second series. Because the block movement step has ordered blocks in tail-sorted order, the first series block in P_i precedes the associated second series block. Swapping the last k records from P_i with the first k records in the second series block would yield an arrangement in which all records in P_i precede those in the second series block. Thus, k is the displacement table entry, E_i .

To find k , each processor with a first series block knows the number of second series records to be searched and the total number of processors that will be accessing the same second series block. (This information is broadcast from the associated second series processor, P_j , as described in § 4.3.3.) An initial second series "midpoint" index is computed based on the processor's position within the other competing processors. This initial index is sent as a request to P_j which returns the indexed key value. The returned key is compared

to the key located in the “midpoint” number of records from the right end of the local first series block. Based on the outcome of this comparison, the endpoints of the search space are adjusted.

When compared, if the local $A[hi-k]$ is less than the midpoint key, we know that k is too large. If that many records were swapped between P_i and the second series block, some small-keyed records would be moved and not all records in P_i would precede the records in P_j . In this case, the high bound of the search is reset to the midpoint. Conversely, if the local $A[hi-k]$ is greater than the midpoint key, there are too few records that would be swapped. Again, not all records in P_i would precede those in P_j after swapping the designated k records due to large-keyed records being left behind in P_i . In this case, the low bound of the search is adjusted to the midpoint.

If the returned midpoint key is equal to the local $A[hi-k]$ key, the high bound is adjusted. In this way, should a future (lower-indexed) midpoint key compare equally to the corresponding local key, moving the high bound ensures that we are narrowing the search for k such that not only is $A[hi-k]$ in P_i greater than or equal to $A[k]$ in P_j , but we also satisfy the condition of $A[hi-(k+1)]$ in P_i being less than $A[k+1]$ in P_j .

After the key comparison and adjustment of a search bound, a new midpoint index is computed and sent to the second series processor as a request. The binary search proceeds in this way until the low and the high bounds are equal. This index value is the displacement table entry for P_i . Such binary searching is able to proceed in parallel with the first and second series processors cooperating in a consumer/producer relationship as described previously.

6.3.1 Timing Results

Table 6.3 lists execution times for Step 3 of the G-L merge on a 32-node CM-5. The second column (Bin/Lin) gives the time for the distributed memory algorithm using both binary and linear searches. The third column (Binary) gives the step execution time under the single binary search method described above. The percent ratio of execution time for the single binary search to the binary/linear search algorithm ranges from 9.6% for 524288 records to 0.3% for 33554432 records. It is anticipated that the percentage difference would continue to increase for larger data set sizes.

Table 6.3: STEP 3 TIMING COMPARISON (ON CM-5)

Number of Records	Time (seconds)	
	Bin/Lin	Binary
524288	0.083	0.008
1048576	0.171	0.009
2097152	0.332	0.010
4194304	0.653	0.011
8388608	1.312	0.010
16777216	2.623	0.016
33554432	5.477	0.014

While the consumer/producer relationship between first and second series processors is still part of Step 3, second series processors handle fewer total requests ($O(\log p)$ rather than $O(n/p)$).

6.4 Combining Block Sorting and Record Movement

As originally outlined in [28], the G-L merge can potentially move each record twice: once in Step 1 and once in Step 4. In this section we develop an innovation on the distributed memory model algorithm that requires only a single movement of records between processors. In this way, the total amount of communication over the entire execution is reduced substantially.

We first show that such an algorithm is possible if we ignore the requirement to make use of only $O(1)$ extra space. This will involve computing the index that a record should be moved into from its original position. This index value is stored in an $O(n)$ array used to permute each record. After this, we show how this same idea can be accomplished with a constant amount of extra storage on each processor without explicitly storing the permutation arrays.

6.4.1 $O(n)$ Extra Memory Version

Theorem 1 *Without regard to space constraints, the two data movement steps (block and record movement) of the G-L merge can be accomplished in a single data movement on an EREW PRAM.*

PROOF. Without loss of generality, we shall assume that the size of the subfiles to be merged is evenly divisible by the number of processors, p , and that each processor stores n/p records. We make the following modifications to the G-L merge algorithm.

After sorting the tail keys in the first step, each processor stores the location of which other block its portion of the data is to be moved. The new position for each record can

be represented as a permutation, π_1 . Such a permutation is implemented as an array of n indices.

Rather than move data between processors to align blocks into tail-sorted order, processors take on the *id* number of that processor to which its block would have been moved according to π_1 . This becomes the *logical processor number* (LPN).

More important than knowing the permutation π_1 , each processor must be able to identify its *left logical neighbor* and *right logical neighbor*. That is, those processors whose logical processor number is one less and one more, respectively, than the local LPN. Appropriate notations for the logical end processors are handled as well. By compiling a complete mapping between processors and their logical processor numbers, each processor can determine its own left and right logical neighbor. The logical processor number along with the left and right logical neighbors is implemented as a doubly linked list of data blocks in tail-sorted order.

Through table look-up and indirect addressing, the logical processor mapping is used for communicating and sharing data. For example, breaker locations (Step 2) and displacement table entries (Step 3) can be computed with the same methods used in the original algorithm under the new processor mapping, rather than the default ordering of processors. Each processor, P_i , computes and stores the displacement table entry for its logical processor. We denote the displacement table entry of the j th logical processor as $E_{[j]}$.

From the displacement table entries, the final destination for each record can be calculated. The record movement step dictates three possible destinations for each first series record: the associated second series block, the right neighbor block or the current position

(no movement). Second series records are to be distributed among the first series blocks or remain in place. Following the convention of [28], we shall use X to denote those first and second series records which do not move from their block during the record movement step, Y to denote the $n/p - E_{[i]}$ first series records that are to be moved, and Z to denote the second series records that are to be distributed among the vacancies left by the displacement of Y records. Our goal is to compute a second permutation, π_2 , to indicate the desired position of each record for the local merge step.

To construct π_2 , rotate records within each first series block to locate the set of Y records in the leftmost portion of the block. The corresponding π_1 array elements are rotated as well. In first series blocks held by second series processors, the Y records will be found to the right of the breaker position after such a rotation.

Movement of X records under π_2 will simply be π_1 . That is, since these records remain at their current location in the block after the record movement step, they need to be moved to the position to which they should have been moved if the block movement had been carried out in Step 1.

The number of Y records in the first series block of logical processor i is $E_{[i]}$. These records are to be moved to the processor that is the right neighbor of the actual processor denoted by the logical processor number (i.e., $LPN + 1$). The destination of these records is computed by adding n/p (the size of the entire data block held by the processor) to each Y record entry in π_1 .

The destination of Z records depends upon the displacement table entries of the asso-

ciated first series block processors. Given that logical processor i holds the leftmost first series block in a series pair sequence, the first $E_{[i]}$ Z records will be moved into the first $E_{[i]}$ positions of the block held by actual processor P_i . These index positions can be computed and stored in the appropriate π_2 elements. The number of Z records to be moved to the block in actual processor P_{i+1} is $(E_{[i+1]} - E_{[i]})$. These records will be placed in the block starting at the $(E_{[i]} + 1)$ position. The third set of $(E_{[i+2]} - E_{[i+1]})$ records will be placed in the actual block starting at the $(E_{[i+1]} + 1)$ position. The remainder of the Z record entries in π_2 are computed in a similar manner.

Under the record movement scheme we can see that at each record position vacated by a Y record, a record from a unique position takes its place. This record is either a Y record from the left neighbor or a Z record from the associated second series block. Similarly, Z records moved are replaced with Y records from the left neighbor's first series block. Thus, π_2 is a permutation of records and is used to move records from their original position to the position necessary to perform the local merge step of the G-L merge.

Hence, through the use of two $O(n)$ permutation arrays and an $O(p)$ processor mapping array, only a single data movement of each record is required to implement the G-L merge on an EREW PRAM. $\square$

The creation and duplication of the processor map defined by the data permutation π_1 is accomplished in $O(\log p)$ steps on the EREW PRAM through pointer jumping techniques. The G-L merge requires a processor to be able to broadcast to the left and right and to identify the associated second series processor to compute the breaker location and displacement table

entry. Once a processor knows the block permutation given by logical processor numbers, the second and third steps of the merge are done in $O(\log n/p + \log p)$ and $O(n/p + \log n)$ steps, respectively. Step 4 computes π_2 and moves the data based on this permutation in $O(n/p)$ time. The final merge requires $O(n/p)$ time. Thus, the total time complexity of the modified algorithm described in Theorem 1 is $O(n/p + \log n)$. This is the same time as the original algorithm in [28].

Corollary 2 *Without regard to space constraints, the two data movement steps (block and record movement) of the G-L merge can be done in a single data movement on a distributed memory model.*

The processor map is done by processors sending their actual and logical processor numbers and logical processor number to each of the other processors in turn. The relevant portions of the permutations π_1 and π_2 are constructed and locally used by processors.

6.4.2 $O(1)$ Extra Memory Version

In Theorem 1, we use the fact that the block movement of the first step and the record movement of the fourth step are permutations of records. In both steps each record is moved to a unique position in the file or else remains in its original position. In this section we demonstrate an algorithmic improvement of the G-L merge for distributed memory machines that uses only a single movement of data in a time-space optimal manner on distributed memory machines. This algorithm can be adapted to shared memory machines by replacing the sending and receiving of messages with coordinated memory accesses and implementing

Step 3 with the phased merge and linear search methods from [28].

We must be able to determine the logical processor order and determine record movements without explicitly storing permutations. As before, we assume that the size of the subfiles to be merged is evenly divisible by the number of processors, p , and that each processor holds n/p records in local memory.

In the first step, after the tails have been merged, each processor P_i knows the processor P_j from where it should expect to receive records if the blocks were to be sorted by the tail key. Upon receipt of the processor *id* number from P_i , P_j uses i as its logical processor number (LPN).

Once the logical processor number is known, a processor must find both its left and right logical neighbors. As defined in the proof of Theorem 1, these are the processors whose logical processor number is one less and one more than the local LPN. The combination of LPN with left and right logical neighbors sets up a doubly-linked list of processors that records the order of tail-sorted blocks.

To find its left logical neighbor, P_i compares its logical number with the logical processor number of P_{i-1} . If the difference between these numbers is 1, then the left logical neighbor of P_i is P_{i-1} . That is, the blocks in P_{i-1} and P_i are from the same sorted list and would have been adjacent had the blocks been moved to their tail-sorted position.

If the difference in logical processor numbers is not 1, a processor holding a block from the other list is the left logical neighbor of P_i . To determine which processor this is, P_i computes the number of blocks (processors) that precede its own block within the same

sorted list from the original data allocation. The difference between this value and the logical processor number of P_i is the number of blocks from the other sorted list that precede P_i in the tail-sorted linked list. By knowing the processor *id* number of the first processor of the other list, P_i can compute the processor *id* of its left logical neighbor. A similar algorithm using the number of processors between P_i and the processors holding the last block of the same and other sorted lists is used to compute right logical neighbors.

The *id* number of the processor holding the first block of the first list will be 0 (zero). The processor holding the first block of the second list can broadcast its *id* number to all other processors in $O(\log p)$ time. The computation of the logical links takes constant time and a constant amount of extra memory at each processor to hold the actual *id* number of the left and right logical neighbors. Thus, the first step of this algorithm takes $O(\log p)$ time for the bitonic merging of the tails and calculation of LPN plus left and right logical neighbors using constant extra space.

The second step finds breaker locations and broadcasts these locations to the left. To determine if a breaker is present within its block, P_i must examine the tail of its left logical neighbor. If this tail is greater than the head in P_i , an internal search is carried out within the block of P_i to locate the breaker position. A parallel segmented scan operation is used to perform the broadcast of breaker locations. By using pointer jumping techniques [65, 14] along the links created in the previous step, this can be done in $O(\log p)$ time and constant space.

The third step, displacement table computation, involves communication between each

processor and its associated second series processor; that is, the nearest logical righthand breaker holder. Each processor knows the actual processor number of this logical second series processor from the broadcasts done in the previous step. Thus, the communication and searching can be done as detailed in § 6.3. Step 3 takes $O(\log n/p)$ time and constant space.

As was done for the proof of Theorem 1, logical processor numbers, displacement table entries and record types are used to determine each record's final block destination. For all blocks, X records will be moved to actual processor P_{LPN} ; for all first series blocks Y records will be moved to the actual processor P_{LPN+1} ; for second series blocks, Z records will be moved to the actual processors denoted by the logical processor numbers of the associated first series block processors.

From the proof of Theorem 1 we know that the final index position (and hence the final processor) for each record can be computed given appropriate displacement table entries and processor *id* numbers. After the internal block rotations, rather than compute the final index positions for all records beforehand, a first approach might be to calculate the index destination for each record, in turn, just before sending that record. By moving records in corresponding positions in parallel, one might assume that records are moved correctly. However, because second series processors contain both second series records and a sub-block of first series records, this approach may lead to a rather complicated pattern of data movement.

The internal rotation of records moves any Y records to the leftmost portion of the first

series block within a second series processor P_j . The first Z record of the associated second series processor is sent out to replace this first Y record. However, since the Y record is not in the actual tail-sorted processor, the Z record must be sent to the actual processor denoted by the LPN of P_j , $P_{LPN[j]}$. The position within the block of records this displaced Z record must occupy will not be the head, where the Z record originates from. Instead it will be a position corresponding to the position of the Y record being replaced. In order to make room for the received Z record, $P_{LPN[j]}$ must compute the final index destination for its own record being replaced and send this record out to the appropriate processor. This destination will again be an internal position within the final block and will require the record currently held within that position to be moved. A troublesome chain reaction of relocating, internally placed records is thus set up.

At best, such a chain of data movements will only replace elements in positions in each block corresponding to the original Y record and take $O(p)$ time. Should another logical Z record be part of the chain, it is possible that a different internal position within each block may be similarly affected. Also, given the high probability that there are multiple second series processors with Z records destined to take the position of internally positioned Y records, data will be moving through multiple chains simultaneously. Since a limit on the number of internal records that must be moved to a single processor at any given time cannot be guaranteed, this multiple and arbitrary data movement cannot be done within constant extra space. The record keeping required at each processor to determine which records have and which have not been previously moved would require $O(n/p)$ space.

To avoid this shortcoming imposed by the original internal rotation scheme, we do not rotate the Y records within the first series block of a second series processor to the lefthand portion of that sub-block. Instead, we simply reverse their order in place. Records are then sent and received in a series of $n/(2k)$ iterations, two records at each iteration: one from the left end and one from the right end of the block of previously unmoved records. Since the destination of a record can be computed at any time, we implement Algorithm T to accomplish the movement of records.

ALGORITHM T:

```

FOR i = 0 to (n/2k)-1 DO
    temp1 = A[i]
    temp2 = A[hi - i]
    send temp1 to computed destination
    send temp2 to computed destination
    receive temp1
    receive temp2
    store temp1 and temp2 in appropriate A[i] and A[hi - i]
END DO

```

To establish that Algorithm T moves records correctly with only constant extra space, we must show that each processor receives exactly two records at each iteration, and that these received records replace those sent out from the processor in the same iteration. There are two cases to be considered.

Case 1. Suppose there are no Y records within any of the first series blocks of second series processors. That is, only X records are in the righthand locations of all processor blocks. The movement of X records simply relocates these records to a (possibly) different processor in the same relative position from which they origin-

ated. Each iteration of Algorithm T permutes a previously unmoved righthand record to the corresponding position in the block held by P_{LPN} . Similarly, X records moved from the lefthand portion of the unmoved list will replace records in the corresponding position of P_{LPN} . The movement of Y records is to corresponding positions of P_{LPN+1} . Z records also replace records in the corresponding positions in the proper processor block dictated by LPNs and displacement table entries. For this first case, at each iteration of Algorithm T, processors send two records and receive two records that replace them.

Case 2. Suppose some second series processors contain Y records to the right of the breaker location. These Y records are at the righthand portion of the block in reverse sorted order. At each iteration of Algorithm T, the rightmost such Y record will replace the leftmost record moved out of P_{LPN+1} while the leftmost unmoved Z record of the associated second series processor will be moved to this Y record's prior location. Other Y records located at the lefthand side of the unmoved portion of each first series processor block will shift down to the actual processor P_{LPN+1} in the corresponding position. The logical left neighbor of the associated logical second series processor will send a record to replace the Z record moved during the iteration. As before, X records move to the same relative positions within (possibly) different blocks. Once there remain no more second series processors with first series Y records left to move, Case 1 is in effect. Again, since we know such data movement to be a permutation, at each iteration

of Algorithm T, processors send two records and receive two records that replace them.

In Case 2 above, since the second series Y records were reversed and moved in the index order of last (lowest key) to first (highest key) into block positions indexed from 0 to $E_{[i]}$, they are returned to sorted order. The Z records that replaced these second series Y records will be in reverse order and must be reversed before the final local merge step. All other X, Y and Z record blocks are in the proper sorted order.

The above modified version of Step 4 moves all records at most once in $O(n/p)$ time and uses only constant extra space. Furthermore, after the above record movement, data is in an order such that the final local merge step may proceed. The total time for the entire parallel merge algorithm is $O(n/p + \log(n/p))$.

From the above description we have the following.

Result 3 *There exists a distributed memory parallel algorithm that merges two sorted lists in $O(n/p + \log(n/p))$ time and $O(p)$ extra space on p processors requiring no more than one movement of each record between processors.*

This algorithm is time-space optimal for $p \leq n/(\log n)$.

6.4.3 Implementation Details

The details of setting up, maintaining and communicating through the derived linking structure of logical neighbors is fairly straightforward. We implemented the passing of records within Algorithm T using message tags. Each processor keeps track of which side of the

block records are being sent, either left or right. This fact is coded in the message tag used to send a record. Upon receipt of the two messages, processors are able to determine at once where to store each of the received records.

Special care must be given to blocks in which the Y or Z sub-blocks are larger than half the block size. For such situations, the index pointers marking the ends of the unmoved portions will encounter these large sub-blocks while the other pointer processes the same type of records. Besides changing the destination processor from that of previous records, these overlaps usually change the destination block side to which such records are to be stored.

In order to speed up the data movement in the record movement step, multiple records should be sent. Before records are moved into message buffers for distribution, each processor determines the maximum number of records that may be sent to a single processor from either side of its unmoved records. This maximum is bounded by the maximum message size constant (BLOCKSIZE) previously set. A global MIN reduction operation on the maximums computes and broadcasts the message size that each processor will use for sending and receiving records during the current iteration of Algorithm T. This global reduction and broadcast will add a factor of $O((n/p) \log p)$ time to Step 4. As we have demonstrated before, in practice this bundling of multiple records into a single data movement can yield dramatic decreases in execution time.

6.4.4 Timing Results

Tables 6.4 and 6.5 list execution times for the G-L merge as originally implemented (Two Moves) and with the communication reduction (One Move) algorithm modifications outlined

above. These timing results are the average of five random test runs generated on a 32-node CM-5. The times in the former Table are for BLOCKSIZE of one record per message while the latter Table presents results of using up to a BLOCKSIZE of 100 records. Figure 6.4 charts the data from Table 6.5.

Table 6.4: SINGLE DATA MOVE TIMING COMPARISON (ON CM-5; BLOCKSIZE = 1)

Number of Records	Time (seconds)	
	Two Moves	One Move
524288	4.052	3.290
1048576	8.700	6.532
2097152	17.994	13.213
4194304	36.358	26.280
8388608	67.172	52.455
16777216	162.103	107.717
33554432	307.608	210.042
67108864	707.767	425.887
134217728	1252.090	856.978

Table 6.5: SINGLE DATA MOVE TIMING COMPARISON (ON CM-5; BLOCKSIZE ≤ 100)

Number of Records	Time (seconds)	
	Two Moves	One Move
524288	0.31	0.28
1048576	0.63	0.50
2097152	1.08	1.04
4194304	1.92	1.80
8388608	4.79	3.58
16777216	10.06	7.56
33554432	20.32	14.92
67108864	41.00	30.77
134217728	83.63	58.95

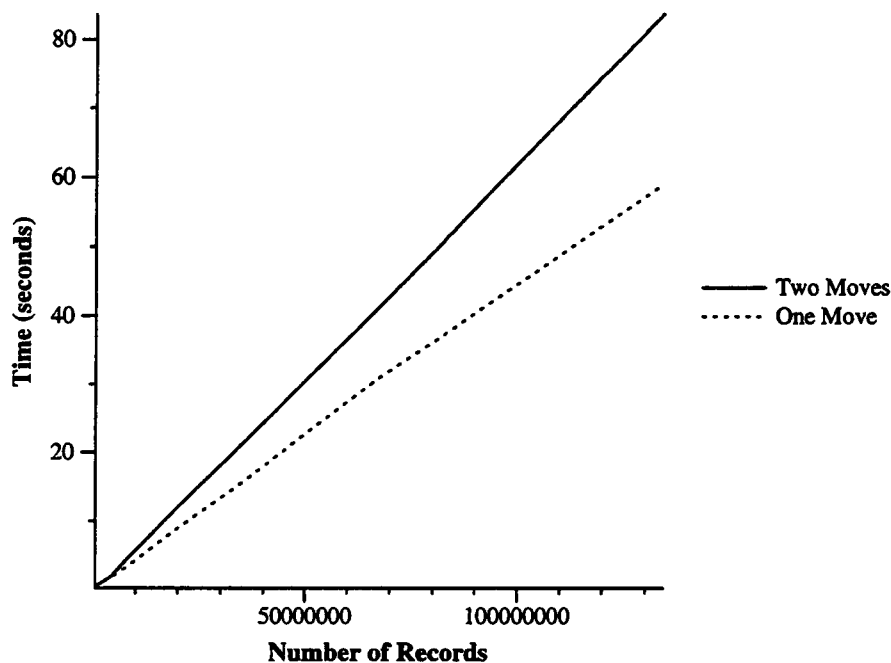


Figure 6.4: ONE MOVE TIME VERSUS TWO MOVE TIME (ON CM-5)

6.5 Summary

In this chapter we presented algorithmic modifications to portions of the G-L merge algorithm suggested and supported by characteristics of different parallel machine architectures and execution paradigms. These changes have been shown to reduce execution times on machines within the target architecture categories (almost 40% in the final merge step on the MP-2; over 99% in Step 3 on the CM-5; and up to 30% overall on the CM-5 when moving records between processors only once).

The code needed for the SIMD modification versions was an additional call to a function that moves data blocks (used in the local merge step) plus the reassignment of index pointers to mark the new ends of blocks to be merged. For the single binary search to implement

Step 3, the complexity of the code was simplified. The single binary search replaced the combination of a binary search to find the tail record's position in the second series block and a linear search to compute the displacement table entry.

Overall, the complexity of the code for our final modification was not increased dramatically. The algorithms for each of the steps remains essentially the same. However, the contents of some messages are expanded, especially when performing scans or broadcasts. Since the ordering of the processors is handled through the logical neighbor indices, our use of pointer jumping necessitates a separate index variable be set aside to reference the actual processor pointed to within the jumping order. This index value is passed back along with relevant data to maintain the structure of the logical neighbor ordering.

Chapter 7

Conclusions and Future Work

7.1 Contributions

We believe that research in non-numeric parallel algorithms is a timely endeavor as platforms for parallel execution become more readily available. These systems will be used to speed up execution and solve a wide range of problems. Specifically, we have examined the process taking non-numeric parallel algorithms originally designed for a single model of computation, and porting them across diverse architectures. This task can present the programmer with a number of challenges, not all of which are predictable from theoretical models of computation. We have used a parallel time-space optimal merge as a prototypical representative of non-numeric computations.

As codes were translated to run on different platforms, new methods of implementation were discovered. These new techniques were often either unsupportable or non-standard on the prior architecture. As a consequence, this dissertation makes the following contributions:

- While the synchronous execution of SIMD machines may be easier to program than

inserting the appropriate synchronizations into algorithms requiring close cooperation between MIMD processors, it is this automatic synchronization that makes SIMD machines unfriendly to non-numeric computations.

- For translating codes from shared memory to distributed memory machines, we have devised a simple technique to identify where message passing must be inserted. While such a straightforward approach may not be the most efficient translation, it does yield a starting point from which a programmer may fine tune an implementation. In some cases, with a little insight and forethought, the number of messages and/or the complexity of the message passing structures may be greatly simplified.
- After converting the same algorithm to many different computers, we realized that such programs could be used as a benchmark kernel for non-numeric parallel algorithms. Our study of the benchmarking literature showed that there did not exist any “bellwether” candidate within the realm of parallel non-numeric computations. Thus, after standardizing the data generation routines for each implementation, we were able to make comparisons of machine performance. One of the side effects of our approach was an ability to load each machine’s memory to capacity and observe its effects on performance. We found that standard numeric benchmarking suites are not very good predictions of non-numeric algorithmic behavior. We advocate the addition of comparison-based, non-numeric codes as part of parallel and supercomputer benchmarks. In fact, since it has been translated to a variety of platforms, the G-L merge itself could be the initial kernel for such an addition.

- Finally, we made several improvements to reduce the amount of communication and computation required on selected programming models. Some of our modifications take advantage of architectural features not present in all execution paradigms. Others exploit the a bottleneck imposed by a combination of the merge algorithm and architectural features. In all cases, our implementations reduced execution time.

7.2 Open Questions

From the results of this research, many open questions beckon including the following.

- *Is there a reduced movement serial in-place merge?* The original G-L merge was based on the serial merge of Huang and Langston [33]. Now that we have found a method of reducing the total number of record movements in the parallel algorithm, it is conceivable that a similar algorithmic modification is possible within the sequential counterpart. Incorporating such a modification would further reduce execution time in the local merge step of the parallel merge.
- *What should a comprehensive parallel non-numeric benchmarking suite include?* Smaller kernels, targeting a single non-numeric operation, will be required. In this way, machine evaluators can easily pick and choose those types of operations that are of the most likely to be executed in practice. Also, complete codes implementing various non-numeric algorithms must be included. Which codes and algorithms to include is a question that would need to be answered by a survey of current industry usage. Certainly, such codes that would be included must be adaptable to a large number of

execution platforms as the G-L merge is.

- *What other problems can make use of in-place techniques?* Algorithms that rely heavily on large-scale data movement, such as the permutation of records, are natural candidates for in-place algorithms. While the efficient use of memory remains a prime concern, investigations into algorithms that make use of time and space optimal methods will continue to provide abundant research opportunities. After Adleman's recent solution of a small combinatorial problem using DNA molecules [2], many researchers have been looking at molecular-based models of parallel computation. We believe that in-place techniques will be a vital part of research into algorithms based on these models.

7.3 Code Availability

The codes described herein are available through the World Wide Web. The URL address is <http://www.cs.utk.edu/~clay/codes.html>.

Bibliography

Bibliography

- [1] C.A. Addison, V.S. Getov, A.J.G. Hey, R.W. Hockney, and I.C. Walton. "The GENESIS Distributed-Memory Benchmarks". In J. J. Dongarra and W. Gentzsch, editors, *Computer Benchmarks*, volume 8 of *Advances in Parallel Computing*. Elsevier Science Publishers B.V., Amsterdam, 1993.
- [2] L. Adleman. "Molecular Computation of Solutions to Combinatorial Problems". *Science*, 266:1021-1024, 1994.
- [3] A. Aggarwal, A. Chandra, and M. Snir. "On Communication Latency in PRAM Computations". In *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures*, pages 11-21, 1989.
- [4] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1989.
- [5] D.H. Bailey, E. Barszcz, L. Dagum, and H.D. Simon. "NAS Parallel Benchmark Results 10-94". Technical Report NAS-94-001, NASA Ames Research Center, October 1994.
- [6] D.H. Bailey, J.T. Barton, T.A. Lasinski, and H.D. Simon (editors). "The NAS Parallel Benchmarks". Technical Report RNR-91-002, NASA Ames Research Center, January 1991.
- [7] K.E. Batcher. "Sorting Networks and Their Application". In *Proceedings of the AFIPS SJCC*, pages 307-314, 1968.
- [8] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.
- [9] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsuing, J. Schwarzmeyer, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. "The PERFECT Club Benchmarks: Effective Performance Evaluation of Computers". *International Journal of Supercomputer Applications*, 3:5-40, 1989.
- [10] L.N. Bhuyan and X. Zhang, editors. *Multi-Processor Performance Measurement and Evaluation*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

- [11] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha. "A Comparison of Sorting Algorithms for the Connection Machine CM-2". In *Proceedings 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3-16, 1991.
- [12] C.P. Breshears and M.A. Langston. "MIMD Versus SIMD Computation: Experience with Non-Numeric Parallel Algorithms". *Parallel Algorithms and Applications*, 2:123-138, 1994.
- [13] R. Cole and O. Zajicek. "The APRAM: Incorporating Asynchrony into the PRAM Model". In *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures*, pages 169-178, 1989.
- [14] T. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.
- [15] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramanian, and T. von Eicken. "LogP: Towards a Realistic Model of Parallel Computation". In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pages 1-12, 1993.
- [16] D.J. DeWitt. "The Wisconsin Benchmark: Past, Present, and Future". In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.
- [17] D.J. DeWitt, D. Friedland, D.K. Hsiao, and M.J. Menon. "A Survey of Parallel Sorting Algorithms". Technical Report OSU-CISRC-TR-81-11, Computer and Information Science Research Center, Ohio State University, December 1981.
- [18] K.M. Dixit. "The SPEC Benchmarks". *Parallel Computing*, 17:1195-1209, 1991.
- [19] J.J. Dongarra. "Performance of Various Computers Using Standard Linear Equations Software". Technical Report CS-89-85, University of Tennessee, Updated May 18, 1994.
- [20] A.G. Ferreira and M.D. Grammatikakis. "Improved Probabilistic Routing on Generalized Hypercubes". In *6th International PARLE Conference Proceedings*, pages 1-12, 1994.
- [21] S. Fortune and J. Wyllie. "Parallelism in Random Access Machines". In *Proceedings of ACM Symposium on Theory of Computing*, pages 114-118, 1978.
- [22] R. Francis and I. Mathieson. "A Benchmark Parallel Sort for Shared Memory Multiprocessors". *IEEE Transactions on Computing*, 37:1619-1626, 1988.
- [23] K. Gallivan, R. Plemmons, and A. Sameh. "Parallel Algorithms for Dense Linear Algebra Computations". *SIAM Review*, 32:54-135, 1990.

- [24] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.
- [25] R.D. Grappel and J.E. Hemenway. "A Tale of Four μ Ps: Benchmarks Quantify Performance". *EDN*, pages 179–265, April 1 1981.
- [26] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.
- [27] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.
- [28] X. Guan and M.A. Langston. "Time-Space Optimal Parallel Merging and Sorting. *IEEE Transactions on Computers*, 40:596–602, 1991.
- [29] M.T. Heath, E. Ng, and B.W. Peyton. "Parallel Algorithms for Sparse Linear Systems". *SIAM Review*, 33:420–460, 1991.
- [30] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1-2):1–170, 1993.
- [31] R.W. Hockney. "A Framework for Benchmark Performance Analysis". In J.J. Dongarra and W. Gentzsch, editors, *Computer Benchmarks*, volume 8 of *Advances in Parallel Computing*. Elsevier Science Publishers B.V., Amsterdam, 1993.
- [32] R.W. Hockney and M. Berry, editors. "Public International Benchmarks for Parallel Computers". *Scientific Programming*, 3(2):101–146, 1994.
- [33] B-C Huang and M.A. Langston. "Practical In-Place Merging". *Communications of the ACM*, 31:348–352, 1988.
- [34] The Industrial Liason Program at University of California at Berkeley. *Eqntott #V9*, 1985.
- [35] M. Jones. Personal communication, May 1996.
- [36] C. Kaklamanis, D. Krizanc, and S. Rao. "Simple Path Selection for Optimal Routing on Processor Arrays". In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 23–30, 1992.
- [37] R.M. Karp and V. Ramachandran. "A Survey of Parallel Algorithms for Shared-Memory Machines". Technical Report 408, (EECS) University of California, Berkeley, 1988.
- [38] P.M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.
- [39] C.P. Kruskal, L. Rudolph, and M. Snir. "A Complexity Theory of Efficient Parallel Algorithms". *Theoretical Computer Science*, 71:95–132, 1990.

- [40] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe. "The Structure of an Advanced Re-targetable Vectorizer". In K. Hwang, editor, *Supercomputers: Design and Applications*. IEEE Computer Society Press, Silver Spring, MD, 1984.
- [41] V. Kumar, A. Grama, G. Anshul, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.
- [42] C.E. Leiserson. "Fat Trees: Universal Networks for Hardware-Efficient Supercomputing". *IEEE Transactions on Computing*, 34:892-901, 1985.
- [43] W.G. Levelt, M.F. Kaashoek, and H.E. Bal. "A Comparison of Two Paradigms for Distributed Shared Memory". *Software: Practice and Experience*, 22:985-1010, 1992.
- [44] M. Li and Y. Yesha. "New Lower Bounds for Parallel Computations". In *Proceedings of 18th ACM Conference on Theory of Computing*, pages 177-187, 1986.
- [45] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Programming Language (ANSI C compatible MPL) User Guide*, Revision: A3, May 1992. Document Part Number 9302-0101.
- [46] J.A. McCann and D.A. Bell. "A Hybrid Benchmarking Model for Database Machine Performance Studies". In J.J. Dongarra and W. Gentzsch, editors, *Computer Benchmarks*, volume 8 of *Advances in Parallel Computing*. Elsevier Science Publishers B.V., Amsterdam, 1993.
- [47] N. Nupairoj and L. Ni. "Performance Evaluation of Some MPI Implementations on Workstation Clusters". In *Proceedings, Scalable Parallel Library Conference*, pages 98-105, 1994.
- [48] C.G. Plaxton. "Load Balancing, Selection and Sorting on the Hypercube". In *Proceedings 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 64-73, 1989.
- [49] U. Ramachandran and M.Y.A. Khalidi. "An Implementation of Distributed Shared Memory". *Software: Practice and Experience*, 21:443-464, 1991.
- [50] O. Serlin. "The History of DebitCredit and the TPC". In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.
- [51] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. "Efficient Algorithms for Parallel Sorting on Mesh Multicomputers". *International Journal of Parallel Programming*, 20:95-131, 1991.
- [52] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1995.

- [53] E. Spertus, S.C. Goldstein, K.E. Schauser, T. von Eiken, D.E. Cutter, and W.J. Dally. "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5". In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 302–313, 1993.
- [54] A. Symvonis. "Routing on Trees". In *Proceedings, Australian Computer Science Communications-17*, pages 297–308, 1994.
- [55] K. Thearling and S. Smith. "An Improved Supercomputer Benchmark". In *Proceedings of Supercomputing '92*, pages 14–19, 1992.
- [56] Thinking Machines Corporation, Cambridge, MA. *C* Programming Guide*, May 1993.
- [57] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, 3.0 edition, May 1993.
- [58] L.G. Valiant. "Parallelism in Comparison Problems". *SIAM Journal of Computing*, 4:348–355, 1975.
- [59] L.G. Valiant. "A Bridging Model for Parallel Computation". *Communications of the ACM*, 33:103–111, August 1990.
- [60] A.J. van der Steen. "The Benchmark of the EuroBen Group". *Parallel Computing*, 17:1211–1221, 1991.
- [61] J.S. Vitter and E.A.M Shriver. "Algorithms for Parallel Memory II: Hierarchical Multilevel Memories". *Algorithmica*, 12:148–169, August-September 1994.
- [62] R.P. Weicker. "Dhrystone: A Synthetic Systems Programming Benchmark". *Communications of the ACM*, 27:1013–1030, 1984.
- [63] R.P. Weicker. "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules". *SIGPLAN Notices*, 9:60–82, 1989.
- [64] R.P. Weicker. "A Detailed Look at Some Popular Benchmarks". *Parallel Computing*, 17:1153–1172, 1991.
- [65] J.C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, 1979.
- [66] W. Zimmermann and W. Löwe. "An Approach to Machine-Independent Parallel Programming". In *Parallel Programming: CONPAR 94-VAPP VI*, number 854 in Lecture Notes in Computer Science, pages 277–288, 1994.

VITA

Clay Breshears was born in Spokane, WA, in 1960. He has been interested in computers since 1976 when he joined a Computer Science Explorer Post and learned to program in COBOL on an IBM 370. After graduating from high school, Clay attended Spokane Falls Community College (AA degree), Eastern Washington University (BS in Computer Science) and Washington State University (MS in Computer Science) in rapid succession. He accepted an Instructor post at Eastern Washington University in the Computer Science Department after completing work for his Master's degree. Clay held that position for four years before returning to school in order to earn a doctorate degree at the University of Tennessee, Knoxville. While in Knoxville, he spent several summers as a Computing and Computer Science instructor with the Tennessee Governor's School for the Sciences and was employed as a Systems Engineer by ApoCom, Inc, during his final year there. Currently, Clay is an Assistant Professor in the Department of Computer Science and Statistics at the University of Southern Mississippi located in Hattiesburg, MS.