# Parallel Benchmarks and Comparison-Based Computing[*]

Clay P. Breshears
Department of Computer Science
University of Southern Mississippi
Hattiesburg, MS 39406
USA

Michael A. Langston
Department of Computer Science
University of Tennessee
Knoxville, TN 37996
USA

## Abstract

Non-numeric algorithms have been largely ignored in parallel benchmarking suites. Prior studies have concentrated mainly on the computational speed of processors within very regular and structured numeric codes. In this paper, we survey the current state of non-numeric benchmark algorithms and investigate the use of in-place merging as a suitable candidate for this role. In-place merging enjoys several important advantages, including the scalability of efficient memory utilization, the generality of comparison-based computing and the representativeness of near-random data access patterns. Experimental results over several families of parallel architectures are presented.

[*]A preliminary version of a portion of this paper was presented at the International Conference on Parallel Computing held in Gent, Belgium, in September, 1995. This research has been supported in part by the National Science Foundation under grant CDA–9115428 and by the Office of Naval Research under contract N00014–90–J–1855.

## 1 Introduction

Only in recent years have non-numeric algorithms been considered for inclusion in parallel and supercomputer benchmarks [2, 8, 22]. 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 [2, 22]. 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, squander an unbounded amount of unnecessary 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 the sequel, we examine the effectiveness of the parallel merge (and hence sort-by-merging) approach first devised in [13]. 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. We also compare these results to those obtained for numeric codes.

In the next section, we survey several known benchmarking efforts, including those for sequential and parallel computational models. In Section 3, we focus on the needs of a suitable non-numeric parallel benchmark. We consider previously-proposed alternatives in this light, and discuss the relative merits of in-place merging. In Section 4, we address implementation details and present timing results, with which we compare different machines. Comparisons are also made to numeric benchmark results. A final section contains a few closing observations and related remarks.

# 2 Previous Efforts

## 2.1 Sequential Benchmarks

We first 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.

### 2.1.1 Dhrystone

The Dhrystone benchmark [26] 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 [27]. The Dhrystone benchmark is intended to measure integer performance on small machines with simple architectures.

### 2.1.2 Stanford Small Programs Benchmark Set

Weicker [28] 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.

### 2.1.3 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.

### 2.1.4 EDN Benchmarks

This collection of programs was developed at Carnegie Mellon University and published by EDN in 1981 [11]. 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.

## 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* [21] make use of a simple quicksort. This algorithm, along with others used for comparison, were written in the parallel language Id90. Francis and Mathieson [10] 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 [5].

Attempts to develop and make available a standardized parallel benchmarking suite are in their infancy. These attempts include the EuroBEN Group [25], the GENESIS Project [1], PERFECT Club Benchmarks [4], NAS Parallel Benchmarks [2] and the vendor-supported SPEC Benchmarks [8]. Recently, a number of researchers interested in benchmarking met and formed the PARKBENCH Committee [14]. The goal of this group is to make available codes that test the capabilities of scalable massively-parallel computers while maintaining strict guidelines on an acceptable benchmarking methodology to ensure that performance evaluations are meaningful across a variety of machines. A majority 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 [2] 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 [22], 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 Connection Machine CM-5 with varying numbers of processors and several data distributions.

Within the integer computation suite, CINT92, of the SPEC Benchmarks [8] 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* [23], that translates logical Boolean expressions into truth table equivalents, makes use of a sorting procedure.

## 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 [7] and the TPC benchmark [20]. 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 [7]. 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 [19] 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 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).

## 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 will employ instead a "bell weather" non-numeric operation: merging. We have selected a parallel, in-place method as our algorithm of choice. An explanation of this selection follows in the next section. Details on the inner-workings of the algorithm itself are contained in an appendix.

## 3 Defining a Suitable Benchmark Algorithm

Gray [12], 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. Not to put too fine a point on it, the goal is to select algorithms that are complex enough to tickle a wide range of architectural features, but not so intricate that the code is indecipherable.

## 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, this simple-minded approach does not scale with limited memory.

## 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; no global scan/reduction operations are involved.

## 3.3 Radix Sort

Parallel radix sort [18] 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 and the receiving processor. 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 [29] are available if one wishes to measure this very limited type of data movement.

## 3.4 Valiant's Parallel Merge

Converting the algorithm of Valiant [24] to a computational model with a fixed number, $k$, of processors is fairly simple. Instead of being able to distribute $\lfloor\sqrt{nm}\rfloor$ processors to merge $\sqrt{n}$ smaller sublists, we merge $O(k)$ lists on the $k$ processors by dividing the first list into $k$ blocks 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 (blocks), 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 second list blocks that are to be merged with the first list blocks. 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 blocks 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.

## 3.5 In-Place Merging

The algorithm of [13] 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 keys, 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. The destination block is based on the sorted position of the last key within each block. 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 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.

## 4 Implementation and Timing Results

Table 1 contains hardware, operating systems, and compiler details on each of the machines we have used within our study.

| | Connection Machine CM-5 (SIMD) | MasPar MP-2 | Connection Machine CM-5 (MIMD) | Intel iPSC/860 | IBM SP2 | Intel Paragon |
|---|---|---|---|---|---|---|
| **Operating System:** | SunOS Release 4.1.2; CMOST Version 7.2 | ULTRIX V4.3 (Rev. MP-3.22) | SunOS Release 4.1.2; CMOST Version 7.2 | iPSC/860 UNIX Sys V 3.2, NX 3.3.2 | AIX 6000 | Paragon OSF/1 Release 1.0.4 Server 1.3 R1_3 |
| **Compiler:** | C∗ Driver Version 7.1 Final Rev f2000 | MPL Version 3.2.14 | SunOS C Compiler with CMMD Extensions | icc/NX Sun4 Rel 4.0 | AIX C Compiler with MPL Extensions | icc/Paragon Version R5.0.1 |
| **Processors:** | SunSPARC1 (plus 4 vector units per node) | proprietary RISC | SunSPARC1 | Intel i860 | RS/6000 (POWER2 architecture) | Intel i860 XP |
| **Memory per Node:** | 32 Mbytes | 64 Kbytes | 32 Mbytes | 8 Mbytes | 128 Mbytes | 16 Mbytes |

Table 1: Machines Used in This Study

### 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.

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.

## 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. Based on the performance metrics outlined in [15], a more useful measure for non-numeric algorithms such as ours is the number of records handled per second.

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 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 file sizes used in our tests are the exact powers of two contained within the ranges shown plus those file sizes midway between successive powers of two. Timings on each machine are taken from the average of five different, random test runs for each file size.

| 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) | 32768 – 3145728 | 0.002 |

Table 2: Average Performance of Each Machine

The graph in Figure 1 depicts the entire set of run times we observed over the course of our experiments. Note that the $x$ axis uses a logarithmic scale.

Memory capacity is another critical measure. We have observed before [6] that memory management schemes can exhibit unexpected behavior at or near capacity. Table 3 lists maximum data memory capacity and corresponding machine performance for three sample

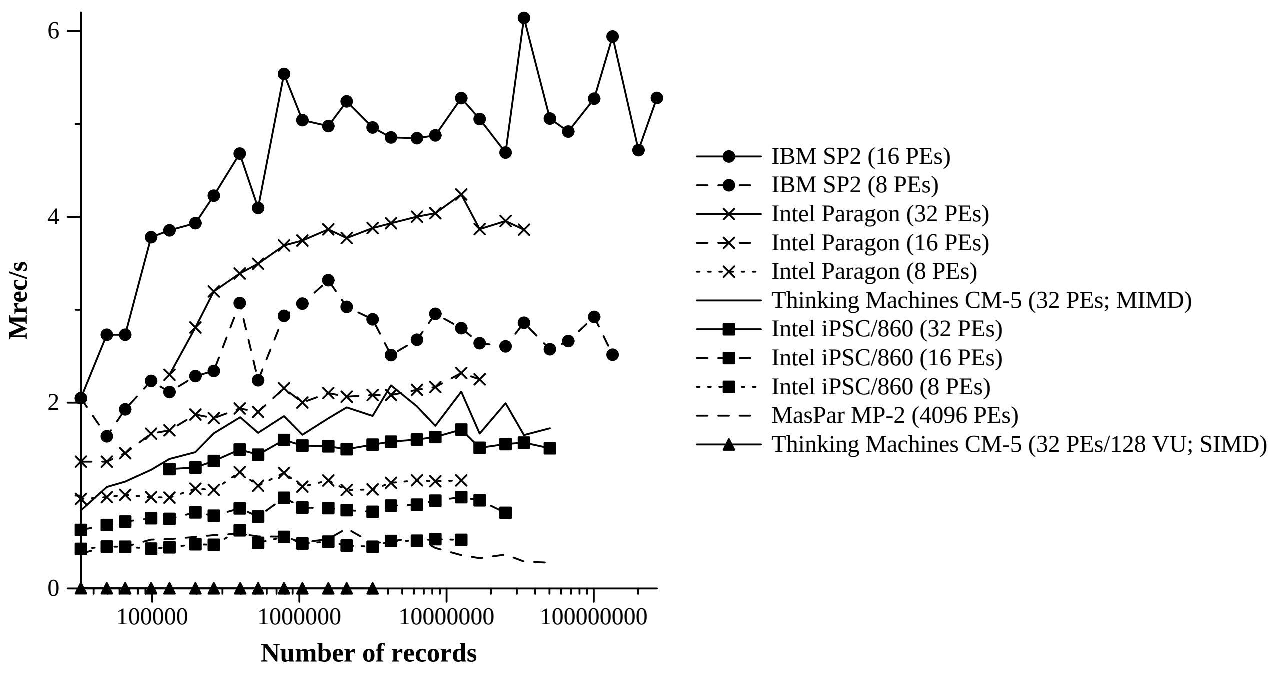

Figure 1: Detailed Performance of Each Machine

configurations: MasPar MP-2 with 4096 processors; Thinking Machines CM-5 (MIMD) with 32 processors; Intel iPSC/860 32 processors.

| 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% |

Table 3: Memory Capacity Measure and Performance

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 [9], 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 results from both benchmarks are shown in Table 4[†]. Numbers in brackets represent relative rankings. Among other things, Tables 2 and 4 both illustrate that SIMD architectures can be extremely unfriendly to non-numeric algorithms.

| Machine | Number of processors | LINPACK $R_{max}$ (Gflop/s) | 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 | 1.9 [1] | 0.001 [3] |

Table 4: Comparison of LINPACK and In-Place Merge Benchmarks

We also note differences between our results and those of the NAS Parallel Benchmark Integer Sort kernel. We have taken from [3] the raw execution times for $2^{23}$ keys and converted them to Mrec/s. See Table 5.

| Machine | Number of processors | NAS IS (Mrec/s) | 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 (OSF1.2) | 32 | 1.074 [3] | 4.039 [2] |

Table 5: Comparison of NAS Integer Sort and In-Place Merge Benchmarks

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.

[†]The 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 algorithm was run.

## 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 resist the temptation to conjecture about 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.

Based on 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.

## References

[1] C. A. Addison, V. S. Getov, A. J. G. Hey, R. W. Hockney and I. C. Walton, "The GENESIS Distributed-Memory Benchmarks," in *Computer Benchmarks (Advances in Parallel Computing 8)*, J. J. Dongarra and W. Gentzsch, eds., Elsevier Science Publishers B.V., Amsterdam, 1993.

[2] D. H. Bailey, J. T. Barton, T. A. Lasinski and H. D. Simon, "The NAS Parallel Benchmarks," RNR Technical Report RNR–91–002, NASA Ames Research Center, January 1991.

[3] D. H. Bailey, E. Barszcz, L. Dagum and H. D. Simon, "NAS Parallel Benchmark Results 10-94," NAS Technical Report NAS–94–001, NASA Ames Research Center, October 1994.

[4] 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. Schwarzmeier, 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 (1989), 5–40.

[5] L. N. Bhuyan and X. Zhang, eds., *Multi-Processor Performance Measurement and Evaluation*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

[6] C. P. Breshears and M. A. Langston, "MIMD Versus SIMD Computation: Experience with Non-Numeric Parallel Algorithms," *Parallel Algorithms and Applications* 2 (1994), 123–138.

[7] D. J. DeWitt, "The Wisconsin Benchmark: Past, Present, and Future," in The Benchmark Handbook for Database and Transaction Processing Systems, J. Gray, editor, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[8] K. M. Dixit, "The SPEC Benchmarks," *Parallel Computing* 17 (1991), 1195–1209.

[9] J. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software," University of Tennessee Technical Report CS-89-85 (updated May 18, 1994).

[10] R. Francis and I. Mathieson, "A Benchmark Parallel Sort for Shared Memory Multiprocessors", *IEEE Transactions on Computing* 37 (1988), 1619–1626.

[11] R. D. Grappel and J. E. Hemenway, "A Tale of Four $\mu$Ps: Benchmarks Quantify Performance," *EDN* (April 1, 1981), 179-265.

[12] J. Gray, The Benchmark Handbook for Database and Transaction Processing Systems, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[13] X. Guan and M. A. Langston, "Time-Space Optimal Parallel Merging and Sorting," *IEEE Transactions on Computers* 40 (1991), 596–602.

[14] R. W. Hockney and M. Berry, eds., "Public International Benchmarks for Parallel Computers," PARKBENCH Committee: Report-1, February 1994.

[15] R. W. Hockney, "A Framework for Benchmark Performance Analysis," in *Computer Benchmarks (Advances in Parallel Computing 8)*, J. J. Dongarra and W. Gentzsch, eds., Elsevier Science Publishers B.V., Amsterdam, 1993.

[16] B-C Huang and M. A. Langston, "Practical In-Place Merging," *Communications of the ACM* 31 (1988), 348–352.

[17] M. A. Kronrod, "An Optimal Ordering Algorithm Without a Field of Operation," *Dok. Akad, Nauk SSSR* 186 (1969), 1256–1258.

[18] 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.

[19] J. A. McCann and D. A. Bell, "A Hybrid Benchmarking Model for Database Machine Performance Studies," in *Computer Benchmarks (Advances in Parallel Computing 8)*, J. J. Dongarra and W. Gentzsch, eds., Elsevier Science Publishers B.V., Amsterdam, 1993.

[20] O. Serlin, "The History of DebitCredit and the TPC," in The Benchmark Handbook for Database and Transaction Processing Systems, J. Gray, editor, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[21] 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," *Proceedings, 20th Annual International Symposium on Computer Architecture* (1993), 302–313.

[22] K. Thearling and S. Smith, "An Improved Supercomputer Benchmark," *Proceedings, Supercomputing* (1992), 14–19.

[23] The Industrial Liaison Program at University of California at Berkeley, Eqntott #V9, released 1985.

[24] L. G. Valiant, "Parallelism in Comparison Problems," *SIAM Journal of Computing* 4 (1975), 348–355.

[25] A. J. van der Steen, "The Benchmark of the EuroBen Group," *Parallel Computing* 17 (1991), 1211–1221.

[26] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM* 27 (1984), 1013–1030.

[27] R. P. Weicker, "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules," *SIGPLAN Notices* 9 (1989), 60–82.

[28] R. P. Weicker, "A Detailed Look at Some Popular Benchmarks," *Parallel Computing* 17 (1991), 1153–1172.

[29] N. Nupairoj and L. Ni, "Performance Evaluation of Some MPI Implementations on Workstation Clusters," *Proceedings, Scalable Parallel Library Conference* 1994, 98–105.

## 6 Appendix: A Brief Overview of In-Place Merging

Fast in-place merging inherently relies on the notions of internal buffering and block rearranging, ideas that can be traced back to the seminal work of Kronrod [17]. Assuming there are $k$ processors available, the data to be merged is viewed as $k$ blocks, each block of size $n/k$, and each block managed by a distinct processor. The parallel method we have employed is from [13]. It will merge two sorted lists of total length $n$ in $O(n/k + \log n)$ time and $O(k)$ extra space on the EREW PRAM model of computation, and is thus time-space optimal for any value of $k \leq n/(\log n)$. It has five main steps.

In the first step, processors sort blocks by their tails (largest-keyed records), then move them accordingly. In the second step, the data is divided into pairs of series. When a processor determines that a series ends in its block, it broadcasts its index to the left using a segmented parallel scan operation. Thus each processor can determine the boundaries of the series around it. In the third step, a special data structure and a phased merge are used so that processors can efficiently determine how many records need to be displaced to their right. In the fourth step, records are distributed as necessary. In the fifth step, a sequential in-place merge [16] is performed on local data.

The algorithm also contains a few subtle special cases. The interested reader is referred to [13] for complete details.