An Improved Supercomputer Sorting Benchmark

Kurt Thearling & Stephen Smith

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142

kurt@think.com / smith@think.com

Abstract: In this paper we propose that the process of sorting be more formally adopted as a performance benchmark for commercial supercomputer applications. To this end we have investigated the use of entropy as a measure of data distribution and propose that it, along with larger datasets, be added to existing sorting benchmarks (such as NAS). Some of the key points in adopting such a benchmark are presented and the results of applying such a benchmark to the CM-5 supercomputer are discussed. As a result of carefully examining this problem, we were able to sort 1 billion 32-bit keys in less than 17 seconds on a 1024 processor CM-5.

1.0 Introduction

The development of commercial markets for supercomputers will depend on the ability of manufacturers to provide performance in areas which are outside the scientific community's traditional needs. Rather than LINPACK and other number-crunching benchmarks, commercial customers will require performance measurements for processing non-numeric data. Tasks such as relational database queries and sorting are examples of the type of processing which will be needed.

We propose that sorting be adopted as an additional benchmark in the repertoire of performance analysis routines. There are two basic features which make sorting a desirable benchmark. First it is simply described as a problem and can be easily scaled in size to provide progressively more difficult benchmarks. (In the work presented here, a maximum of 2^{30} 32-bit keys were sorted, requiring a minimum of 4 Gbytes of memory). Second, sorting requires that massive amounts of data be communicated between processors. In both shared and distributed memory machines, the ability to move data efficiently will dictate performance in many commercial applications. Sorting could be considered the prototypical benchmark of data movement performance (which is closely related to the communication bandwidth in a distributed memory machine) without having to create a contrived example.

As will be described later, it is important to correctly characterize the distribution of the data that is being sorted. We have decided to use an information theoretic measure which characterizes the distribution of keys in terms of the entropy of the key values. As we will show, the sorting techniques described in this paper are robust to changes in the distribution of the key values.

2.0 Sorting as a Benchmark

Sorting has recently been proposed as an important benchmark kernel for the NAS parallel benchmarks [Bailey 91]. These benchmarks have been produced in an effort to improve upon the evaluation of large parallel supercomputers that were being provided by the Livermore Loops [McMahon 86], LINPACK [Dongarra 88a, 88b] and PERFECT club [Berry 89] benchmarks. The intent of NAS was to provide a benchmark that was reliably implemented from a simple "paper and pencil" description while allowing for problem sizes and algorithmic modifications that more fairly reflected problems and kernels that were of interest on today's parallel supercomputers. Specifically, the NAS parallel sorting benchmark consists of sorting 8 million 19-bit

1063-9535/92 $3.00 © 1992 IEEE

integers produced from the average of 4 randomly generated numbers between 0 and 2^{19}.

While these new benchmarks are welcome additions they, perhaps, do not go far enough. Specifically there are parallel supercomputers which are today capable of sorting 1 billion keys in memory in under a minute [Baber 91] and thus the 8 million keys of the current NAS benchmark is relatively small. Additionally the NAS benchmark allows for only a single data distribution that is only moderately non-uniform (the distribution created by the sum of 4 random values). Many real world data distributions are far less uniform than this distribution and some are more uniform. It will be important to determine the performance of sorting systems and algorithms over a wide range of distributions as some algorithms can take advantage of data distributions that are known a priori to be uniform [Baber 91].

3.0 Characterizing the Keys

While the performance of some sorting algorithms, such as bitonic sort, are unaffected by the distribution of the data being sorted, some of the currently most useful algorithms (e.g., samplesort, block radix) are dependent on the data distribution [Blelloch 90]. Algorithms which are not robust to variations in the distribution of sorting keys can have performance problems when they encounter non-uniform data. In the work presented in [Baber 91], the sorting algorithm relies on the fact that the keys are uniformly distributed to insure proper load balancing. If the keys were not uniformly distributed (say they were a list of customers' ages), the majority of processors would be idle while the minority would be overloaded. The goal of our work was to develop a *simple* metric which would allow for the characterization of the key distribution.

It should be noted that there are two possible interpretations of the word "distribution." The first refers to the probability distribution of the values of the keys (e.g. Are low-valued keys more common than high-valued keys?). The second interpretation refers to the way in which the keys are physically placed initially in the memory (e.g. Are the keys already in sorted order? Are they in reverse sorted order?). In this paper we are referring to the first of these two interpretations.

For N 32-bit keys, there are $\binom{2^{32}+N-1}{2^{32}-1}$ possible key distributions [Knuth 68]. If there are one billion (2^{30}) keys, this number is 10 E 1166738659. Obviously it would be impossible to characterize the sorting performance over any but a very small subset of these possibilities.

One technique which has often been used to characterize the distribution of data is entropy measurement. The Shannon entropy [Shannon49] of a distribution is defined as

$$-\sum p_i \cdot \log p_i$$

where p_i is the probability associated with key i. If the logarithm is base 2, the entropy of the key distribution specifies the number of unique bits in the key. For example, if every key had the same value (say 927), the entropy of the key distribution would be 0 bits. On the other hand, if every possible 32-bit key were represented the same number of times (i.e., a uniform distribution), the entropy of the keys would be 32 bits. In between these two extremes are entropies of intermediate values.

In many real world databases there will be fewer bits of entropy for a distribution than bits in the data structure representing the key. Customer account numbers are a good example of this. Often not all possible account numbers are used or it may be the case that certain prefix digits are used to organize the data. For example, a leading order digit of 1 in an account number might specify commercial customers while a leading order digit of 2 might specify individuals. No other leading order digits are allowed. Assuming an eight bit character representation of the digits, the 8 bits in the character are used to represent a 1 bit quantity.

The goal of this work is to evaluate sorting algorithms as the entropy of the key data is varied. To evaluate an algorithm, it is necessary to either measure the entropy of a test set or generate a test set with a specified entropy. We have chosen to generate key data which spans a range of entropy values. To accomplish this, there are many possible algorithms. One technique would be to simply take a uniform set of keys with 32 bits of entropy and zero out the leading order N bits. This would generate keys with (32 - N) bits of entropy. A more interesting technique would produce keys whose individual bits are between 0 and 1 bit of entropy. We propose one such technique here. Unquestionably there are other techniques which could also perform this task. We believe the technique we have developed is general enough to serve the desired purpose.

The basic idea is to combine multiple keys having a uniform distribution into a single key whose distribution is non-uniform. The combination operation we will be using is the binary AND. For example, take two 32-bit keys generated using a uniform distribution (we will assume that the individual bits are independent and that the two keys are independent). In this case, each bit of the keys will have a .50/.50 chance of being either a zero or a one. If we AND these two keys together, each bit will now be three times as likely to be a zero as a one (.75/.25). This produces an entropy of .811 bits per binary digit for a total of 25.95 bits for the entire key (out of a possible 32 bits). If we repeat this process using additional uniform keys, the entropies of the key distributions continue to decrease:

Number of Keys ANDed Together	Entropy
1	1.0 bits
2	.811 bits
3	.544 bits
4	.337 bits
5	.201 bits
Infinite	0.0 bits

The difference between successive ANDings is approximately twenty percent of the total for the first five ANDings. It then takes an infinite amount of additional work to decrease the entropy completely to zero.

In the results presented in this paper, we have characterized the algorithm performance for key entropies of 0, 6.42, 10.78, 17.41, 25.95, and 32 bits. This corresponds to data that has been set to a constant value, the first four ANDings, and uniform data.

4.0 Radix Sorting

A radix sorting algorithm can be implemented to exploit known uniform data distributions and has recently been done so with high performance [Baber 91]. A bucket sort is another, and probably more efficient algorithm, for known uniform data distributions [Cormen 90] as it can be accomplished with only a single interprocessor communications step and a local sort as opposed to the numerous interprocessor communication steps of the radix algorithm.

The radix algorithm is, however, of interest to us as it has also been implemented with high performance for data sets with unknown (non-uniform) distributions [Blelloch 91]. We will then utilize the radix algorithm as described in [Blelloch 91] to see its performance over a variety of different data distributions.

In brief the radix algorithm makes multiple passes over the keys sorting on a subset of the total bits of the keys and moving from the least significant to most significant bits with each pass. Each pass uses a counting sort which consists of three basic phases:

1. Build a histogram of key values locally on each processor.
2. Count the number of occurrences of each histogram row across processors.
3. Use these counted values to determine the new address of the key. Then permute the keys based on this new address.

Histogram Generation: In this phase, each key is examined (in order) and the associated histogram bucket is incremented. For an R bit radix, there are 2^R buckets. The primary limitation on performance is the speed at which memory can be accessed.

Our first attempt at implementing this phase was used a naive, simple approach:

```
Index = Shift and Mask Key
Histogram[Index]++
```

An important factor in the performance of the histogramming phase is the cache. In the case of the CM-5 processing nodes (SPARC processors), a cache line holds eight 32-bit words. Therefore, when the keys are examined, the first key in a line misses. If that line is not quickly replaced (simulations show that it isn't), the next seven key references are all cache hits. Once each key has been examined it is not used again in this phase. On the other hand, the histogram entries are accessed randomly. When the histogram is at least as big as the cache, we have found that once the transients have died out, most of the cache contains the histogram.

A simple improvement to this approach was then suggested to us [Culler 92]. In our original implementation, a 16-bit blocksize was found to be optimal. This means that 2^{16} histogram buckets are necessary. Since there are 16K 32-bit words in the cache, only 25% of the histogram buckets can be in the cache at any instant if each bucket is represented as a 32-bit integer. Instead of using 32 bits for each bucket, we changed each bucket's representation to 8 bits. This allowed us to fit the entire histo-

gram into the cache. After each bucket is incremented, the result is checked to see if there was an overflow. If an overflow did occur, an auxiliary bucket (the high-order bits for the bucket) was incremented. Since an overflow does not happen very often (on average it occurs once every 256 increments), nearly all of the 8-bit histogram buckets will be available in cache. When the process is completed, the complete histogram buckets are reconstructed by combining the 8-bit buckets and their corresponding auxiliary bucket. This "trick" resulted in a 37% improvement in the histogram generation time.

Scanning the Histogram: Once the histogram entries have been updated, the results need to be communicated to other processors. A parallel prefix plus operator ("plus scan") is applied to each one of the histogram entries in parallel. Although this requires 2^R scans, the CM-5 is designed to do this efficiently. The actual implementation pipelines this operation, and as will be shown later, the cost of this phase is relatively small.

After the plus scan has been performed, the results in the last processor must be copied back to each processor. This is also performed using a scan operation ("max-scan") using the CM-5's global control network. This operation is also pipelined to take advantage of the long histogram arrays.

Finally, a local plus scan is used to update each of the processors local histogram entries. Once this is accomplished, each histogram entry will contain the absolute address for the first local key associated with that histogram index.

Sending the Keys: After the histograms have been scanned, the resulting entries will contain the destination addresses for the keys. Since they are absolute addresses, we must first separate them into processor and array indices. If there are 2^J processors and 2^K keys per processor, the low order K bits specify the array index. The next J bits then specify the processor index. If the number of keys per processor is a power of two, the address generation can be performed using shift and mask operations. Otherwise, modulus and divide operators are used. When the key is sent to the destination processor, the array index of the destination is appended to the key message packet.

5.0 Performance Figures for the CM-5

We have benchmarked our algorithms on CM-5 configurations ranging from 4 to 1024 processors [Thinking Machines 91]. Our first prototype of the algorithm was running within 24 hours of starting this project (at approximately 80% of the performance reported in this paper). It should also be noted that the results presented here are for CM-5 systems without the currently available vector units. Additional results showing the performance improvements when the vector units are used will be submitted for publication in the near future.

We will first discuss our results using a very large number of uniform (entropy of 32 bits) keys on a 1024 processor machine. Later we will discuss our results as the entropy of the key values is varied. In our first experiment, the goal was to sort one billion (2^{30}) 32-bit keys as quickly as possible. Each processor was allocated one million keys. Since both the number of passes and the complexity of each pass is determined by the radix size, we have experimented with various radix sizes. The number of passes is equal to $\left\lceil \frac{32}{R} \right\rceil$. The complexity of the histogram generation and key sending are not directly dependent on the size of the radix (but there are some indirect effects on the cache). Thus there will be three passes with an 11 bit radix and two passes with a 16 bit radix. It should be noted that a 12 (or 13, 14, or 15) bit radix is no better than a 11 bit radix in terms of the number of passes. All require three passes. Although the number of passes does not decrease in these cases, the complexity of each pass does increase. The number of scans is exponentially proportional to the radix. One additional bit in the radix doubles the number of histogram entries and thus the number of scans is also doubled.

We have determined that for large numbers of keys (per processor), it is better to use a longer (16 bit) radix. The total sorting time is dominated by the send and the effect of the scan times is relatively minimal. Increasing the scan time by a factor of 32 (when going from an 11 bit to a 16 bit radix) is more than compensated for by reducing the number of sends. The general relationship between the total sort time, the number of keys per processor, and the size of the radix is described by the following equation:

$$T = \left\lceil \frac{32}{R} \right\rceil \times [2^R \cdot t_{scan} + N \cdot (t_{hist} + t_{send})]$$

where T is the total time for the sort, N is the total number of keys per processor, R is the radix, t_{hist} is the histogramming time (per key), t_{scan} is the scan time (per histogram entry), and t_{send} is the time to send one key to an arbitrary processor (including any additional overhead sent with the key). In addition to the factors includ-

ed in this equation, there are non-linear performance factors involving the cache which influence the timings as the radix is varied.

The total time taken by each phase is listed below. Since the radix was 16 bits, these times are the sum of two passes of the algorithm:

Histogramming	2.0 seconds
Scanning	0.5
Sending	14.2
Total	16.7 seconds

This corresponds to 64.3 MSOPS (millions of sorting operations per second).

These same experiments were also run on a 64 processor CM-5 (again with one million keys per processor) to determine the performance as the number of processors was varied. Since the histogramming phase is entirely local, there was no change in performance for that phase. The scan and send times did vary slightly as interconnection network increased in size. The observed performance was within specifications.

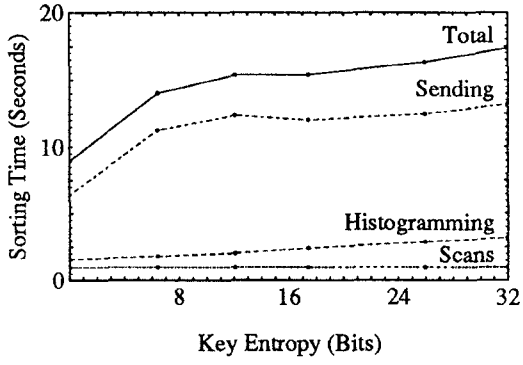

Figure 1: Sort Time versus Key Entropy for a 64 processor system with 1 Million keys per processor.

The performance of the sort for various key entropies is illustrated in Figure 1. As can be seen, decreasing the entropy actually decreases the sorting time. This is due primarily to an increase in cache reference locality as the key distribution becomes more skewed (i.e., has less entropy). In some sense the algorithm is able to take advantage of the structure in the key distribution and the performance increases. In addition, there is a fairly substantial jump in the performance when the entropy of the key distribution falls all the way to zero. This is due to the fact that in this case the data is already in sorted order (since all the keys have the same value, they are already sorted) and don't need to be moved. In that case no global communication is necessary for the send.

6.0 Conclusions

We have proposed that sorting is an important benchmark for both scientific and commercial applications of parallel supercomputers and have shown that further variety in the size and distribution of sorted data is now necessary. To this end we have proposed that the maximum data set sizes increase to the level of a billion or more keys (sizes that are currently being sorted by at least two supercomputers) and that consideration of the data distribution be added to the existing NAS sorting benchmark. This second extension of the current NAS sorting benchmark seeks to cleanly and simply identify the distribution of data being sorted via an entropy measure. This measure will more fairly characterize the uniformity of different data distributions and hopefully provide insights into which parallel algorithms and architectures are the best match for particular problems. Though this entropy measure does not take into consideration such variables as the distribution of data on the processors or sorting stability it is, nonetheless, a step in the right direction that is easily included in the standard benchmarks. We hope that it will be in the future.

7.0 Acknowledgments

The authors would like to thank Mark Bromley, Steve Heller, Marco Zagha, and Guy Blelloch for their assistance with the work presented in this paper.

8.0 References

[Baber 91] M. Baber. An Implementation of the Radix Sorting Algorithm on the Touchstone Delta Prototype. *Proceedings of the Sixth Distributed Memory Computing Conference*. Portland Oregon, IEEE Press, May 1991.

[Bailey 91] D. Bailey et al. The NAS Parallel Benchmarks - Summary and Preliminary Results. *Proceedings of SuperComputing* 158-165, November 1991.

[Berry 89] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3:5 - 40, 1989.

[Blelloch 90] G. Blelloch et al. A Comparison of Sorting Algorithms for the Connection Machine CM-2. *Symposium on Parallel Algorithms and Architectures*, Hilton Head, SC. 3-16, July 1991.

[Culler 92] D. Culler, Personal communication (via Steve Heller), University of California, Berkeley.

[Cormen 90] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. MIT Press. 1990.

[Dongarra 88a] J. Dongarra. The LINPACK Benchmark: An Explanation. *SuperComputing* 10-14, Spring 1988.

[Dongarra 88b] J. Dongarra. *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment*. Technical Report MCSRD-23, Argonne National Laboratory, March 1988.

[Knuth 68] D. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley: Reading, MA, 1968.

[McMahon 86] F. McMahon. *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*. Technical Report UCRL - 53745, Lawrence Livermore National Laboratory, Livermore California, December 1986.

[Shannon 49] C. Shannon and W. Weaver, *The Mathematical Theory of Communication*, University of Illinois Press: Urbana, 1949.

[Thinking Machines 91] CM-5 Technical Summary, Thinking Machines Corporation, October 1991.