An Evaluation of Sorting as a Supercomputer Benchmark (preliminary version)

Abstract:

We propose that sorting be considered an important benchmark for both scientific and commercial applications of supercomputers. The purpose of a supercomputer benchmark is to exercise various system components in an effort to measure important performance characteristics. In the past numerous benchmarks have been defined in an effort to measure the performance issues associated with numeric computing. These benchmarks stressed arithmetic operations (in particular, floating-point arithmetic). In recent years supercomputers manufactures have started to look closer at non-numeric processing tasks, such as databases and information retrieval. The ability to operate on large amounts of non-numeric data will be crucial in the future. This paper discusses the appropriateness of sorting as a benchmark for non-numeric computing tasks. The paper describes previous work in this area and defines a set of architecture independent sorting benchmarks.

Contact: Kurt Thearling
phone: (617) 234-1000, fax: (617) 234-4444
kurt@think.com

1 Introduction

Although sorting is one of the most studied problems in computer science, it has received relatively little attention in the field of supercomputing. Traditional vector supercomputers have been used primarily for numerical analysis and the processing of large, regular arrays. Sorting, on the other hand, is associated with non-numeric applications that typically offer little vectorization and traditionally have been implemented on scalar computers. The newer class of parallel supercomputers, however, do not rely as heavily on vectorization to achieve speedup. As such they have become attractive platforms for very large scale non-numeric applications capable of solving problems that previously were considered intractable. It seems then that the inclusion of sorting as a supercomputer benchmark is both timely and appropriate. In this paper we propose a formal, architecture independent description of sorting applications and introduce a set of specific benchmark cases with which sorting performance can be fairly evaluated. Our hope is that this work will be of benefit in evaluating both sorting algorithms and the computer architectures that they are implemented on.

There are several features of sorting which make it a desirable benchmark. First, it is simply described and well known as a problem; second, it can be easily scaled in size to provide progressively more difficult benchmarks, and lastly, by its nature, it exercises a system's ability to compare and move large amounts of data — often the most expensive portion of any scientific or commercial application. In a parallel processing system, this ability to move data corresponds to the bisection bandwidth of the system [22]. 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 without having to create a contrived example.

In addition to overall speed there are many issues involved in defining a sorting benchmark, including: stability, determinism, memory efficiency, load balancing of the sorted data and the difficulties in extending the sort to perform a rank or to sort complex non-integer keys. In this paper we use the above six descriptors to define the sort being used and then evaluate its performance along four different dimensions: number of keys, key size, distribution of key values and initial allocation of data to memory.

For these benchmarks we have focused on the likely sorting applications that would be performed on supercomputers available today or in the near future. In general this means that we have focused on relatively large sorting problems of approximately 100,000 to over 10 billion keys that range in size from 8 to 256 bits. This seems to characterize the spread of sorting cases encountered today in both scientific and commercial applications, and pushes just slightly into what we can expect supercomputers to be capable of in the near future. Even an in-memory sort of 10 billion 256 bit keys should be possible soon with technological advances in memory capacity. The current generation of supercomputers are being built with tens to hundreds of gigabytes per system and in

the near future we will undoubtably see a supercomputer with a terabyte of RAM. We are today not that far away as a number of recently published results describe where at least 1 billion keys have been sorted on existing supercomputers [3, 37].

This paper is broken up into four main sections. In the next section we review previous work in using sorting as a benchmark. In the second section we give an overview of some of the most common internal sorting implementations and provide a high level taxonomy, breaking the various systems into counting sorts (e.g. radix), fixed-topology sorts (e.g. bitonic), and partitioning sorts (e.g. sample sort). The fourth section of our paper formalizes the six descriptors that define a "sort" and introduces four parameters that when varied can have significant impact on sorting performance. The fifth section introduces a set of specific benchmarks helpful in determining the performance of a sorting implementation over a wide range of variations along these four dimensions.

2 Previous Work

Benchmarks can be considered useful for comparing computer performance only when the benchmark problems are representative of the workload. Because supercomputers traditionally have been employed for numerical analysis, supercomputer benchmarks have emphasized numerical algorithms. None of the early benchmarks, including the Livermore Fortran Kernels [26], the Linpack Benchmark [14], the original NAS Kernels [4], or the PERFECT Club [7] have included sorting as a benchmark. All of these benchmarks, however, were conceived and designed primarily for vector supercomputers, (although recently some results from distributed memory parallel computers have been reported). The only existing supercomputer benchmark designed from the outset for parallel computers is the NAS Parallel Benchmarks [5]. It is significant that this benchmark not only is the most recent, but also is the only one to include a sorting kernel.

Although not the first to suggest sorting as a benchmark for parallel computers, the NAS Parallel Benchmarks represent the first instance where a sorting benchmark has received widespread acceptance by the supercomputing community. Sorting as a parallel computer benchmark was first suggested by Francis and Mathieson [15]. However, the primary goal of that work was to present a parallel merge algorithm with practical application to sorting on a shared memory multiprocessor, and not to outline a sorting benchmark for parallel computers. Sorting on vector computers was in effect used as a benchmark by Rönsch and Strauss [32], where the sorting performance of several Amdahl and Cray systems was compared. As a benchmark, however, that work met with limited acceptance and performance figures for only two other systems (IBM [10] and ETA [27]) were reported. The problem considered for that work was that of sorting N random numbers uniformly distributed in the interval (0,1) using seven different sorting algorithms. Francis and Mathieson also had suggested uniformly distributed random numbers for their sorting benchmark. Unfortunately, few real world data distributions are uniform, and the sorting performance observed on uniformly distributed

data generally will not be representative of the performance achieved on non-uniform data distributions. This is especially true on parallel machines where, at least for some sorting algorithms, non-uniformity in the data will lead to poor load balance and consequently poor performance.

The sorting kernel in the NAS Parallel Benchmarks attempts to overcome this deficiency by purposely specifying a non-uniform data distribution created as the average of 4 random numbers in the interval $(0, 2^{19}]$. The result is an approximate Gaussian distribution with variance $2^{32}/3$. The variance can be decreased by increasing the number of random numbers averaged; however just the one distribution is considered for the benchmark.

There are at least three criticisms one can make of this kernel from the point of view of establishing a general sorting benchmark. This kernel was originally proposed because of its significance in parallel implementations of Monte Carlo simulations of neutral gases where only integer sorting is required. For this reason, it is *not* a general sorting benchmark, but a small integer sorting benchmark. In other words, the benchmark sorts integers in the restricted range $(0, 2^{19}]$ rather than the full word range $(0, 2^{32}]$. This restriction on the range would not be an important criticism however, if the benchmark did not also allow unstable sorting. An efficient *stable* small integer sort can easily be used in building a radix sort. For a stable sort, one can extrapolate the performance of a machine for sorting on the full range given the performance on the restricted range. This, however, is not true for unstable sorts. The third and most relevant criticism is that just a single distribution with only moderately non-uniform data is considered. In real world applications the data distributions can have much greater non-uniformity, and to accurately gauge the performance of a system for general sorting problems one would like to benchmark it over a wide range of distributions. This paper describes a methodology for generating distributions of arbitrary non-uniformity to be used for a general sorting benchmark.

3 Sorting Algorithms

Since there are so many diverse approaches to sorting, hundreds of sorting algorithms have been proposed, for both serial and parallel machines. This section reviews some of the most practical parallel sorting algorithms, focusing on algorithms that have already been efficiently implemented on supercomputers. (For a broader treatment of parallel sorting, see the surveys by Akl [1] and Richards [31]). The purpose of the section is to outline the current state of the art in practical parallel sorting algorithms so that these might be referenced when looking for an efficient sorting algorithm for a new machine. The sorting algorithms we consider can be categorized into three general classes: *counting-based* sorts, *fixed-topology* sorts, and *partitioning* sorts.

3.1 Counting-based sorts

Counting-based sorts work by treating keys as integers in the range $\langle 0 \ldots (m-1) \rangle$. Unlike *comparison-based* sorts such as quicksort [18], counting-based sorts determine the ordering of keys by counting the number of occurrences of each possible value, rather than by comparing pairs of keys. Counting sorts are an attractive alternative to comparison-based sorts since for n keys they run in $O(n)$ instead of $O(n \lg n)$ time.

Stable counting sorts are important as building blocks for *radix* sorts, which are used for sorting integers that are too large to be sorted in a single application of a counting sort. Radix-based sorts work by breaking keys into digits and sorting one digit at a time using a counting sort. For example, a 32-bit integer could be treated as four 8-bit digits. The digit size is usually chosen to minimize the running time and is highly dependent on the implementation and the number of keys being sorted. The most common version of radix sort starts from the *least* significant digit and works only if the ordering generated in previous passes is preserved. In that case the counting sort must be stable.

One way to parallelize counting sort is to assign a different range of keys to each processor. For example, Baber's radix sort for Intel Touchstone Delta [3] performs a counting sort on values in the range $\langle 0 \ldots (p-1) \rangle$ by assigning one bucket to each of the p processors and sending all keys with value i to processor i. While this algorithm works well for uniformly distributed keys, non-uniform distributions can cause a severe degradation in performance, and in the worst case, the algorithm can exceed the available memory. A similar approach to parallelizing counting sort, Dagum's *queue-sort* [13] for the Connection Machine CM-2, uses a fixed amount of memory for any distribution, but has a running time that depends heavily on the distribution. Furthermore, queue-sort is not stable because it is based on a parallel communication primitive that enqueues messages in an unspecified order. Thus it can not be used to build a radix sort. However, queue-sort is efficient for its intended application to particle simulation.

Another approach to parallelizing radix sort assigns a separate set of m buckets to *each* processor, allowing each processor to compute a histogram using only local computation. The histograms are then combined using parallel summing operations [20]. This parallel radix sort algorithm has been efficiently implemented on the CM-2 [8], CM-5 [37], and Cray Y-MP [43]. This algorithm has the advantages that it is stable, and that the time to compute the rank of the keys does not depend on the distribution. (However, on some machines the time to permute the keys based on the rank depends on the permutation. See Manzini's version of radix sort [25] for a version of radix sort that is completely distribution-independent.)

There are two minor disadvantages to radix sort: it does not perform well with large keys, since the running time is proportional to the key size, and it can not be executed in place (i.e. with no temporary memory) . However, radix sort has several advantages over other sorting algorithms. It is simple to implement, deterministic, load-balanced, stable, fast for short keys, and fairly efficient

over a wide range of problem sizes. Furthermore, a radix-based *rank* operation can be implemented at no additional cost compared to a radix sort.

3.2 Fixed-topology sorts

Fixed-topology sorting algorithms are algorithms that use a fixed interconnection network between the processors, such as a hypercube or a grid, and that require no data-dependent communication patterns.

The earliest and most famous of the fixed-topology sorts is Batcher's bitonic sorting network [6][1]. There have been several implementations of Batcher's bitonic sort on parallel machines. These include an implementation for the CM-2 [9], the Carnegie Mellon/Intel iWarp [36] and the Maspar MP-1 [28, 17] as well as several others [34, 32]. If there are multiple keys per processor and a sequential merge is used after each communication, then the asymptotic running time for sorting n keys on a p processor hypercube is $O((n/p)(\lg n + \lg^2 p))$ [20] and on a 2-dimensional grid it is $O((n/p)(\lg n + \sqrt{p}))$ [38]. Because of the small constant in the algorithm and the simple and oblivious communication pattern, the sort is quite efficient on most parallel machines, and it is often used as the sort to which other sorts are compared.

In addition to Bitonic sort, there are several other sorting algorithms that have oblivious routing patterns. Out of these both columnsort [23] and smoothsort [12] are reasonably practical when the number of keys is much larger than the number of processors (for p processors, columnsort requires p^3 keys to run most efficiently). Columnsort has been implemented on the CM-5 with running times that were not as fast as some of other sorts on the CM-5 (including radix and sample sorts), but which for a large number of keys were within a factor of 2 of the best running times [40]. Considering that the CM-5 is not a hypercube, for which the sort is designed, this is a reasonably good performance.

The main advantages of fixed topology sorts is that their communication performance is oblivious to the distribution of the keys, and they are well suited for direct implementation on machines that don't efficiently support dynamic or irregular communication patterns. The disadvantage is that on machines that do have efficient point-to-point communication, the fixed-topology sorts can require more communication than other sorts. An additional advantage of bitonic sort is that it can be executed in place requiring no additional memory. However, this prohibits the use of local merges, making the running time $O((n/p)\lg^2 n)$.

[1] Batcher also suggested a related sorting network called an odd-even merging network, but because it is not as composable as the bitonic network, it has not been used as much in practice

3.3 Partitioning sorts

Partitioning sorting algorithms select a subset of the keys that partition the data and then use these partition elements to route keys to separate sets of processors. There are two main subcategories of partitioning sorts: parallel quicksorts, and sample sorts.

Several parallel variations of quicksort have been suggested, each of which uses a single key at each level of the recursion to partition the data. The simplest variation runs the recursive calls to quicksort in parallel [33]. This variation does not offer very much parallelism since only a single processor is used for the initial partitioning: this partitioning requires $O(n)$ time, so one can only expect an $O(\lg n)$ speedup over the serial algorithm. Wagar suggested a variation called hyperquicksort [39] that does the partitioning in parallel. The algorithm uses a hypercube connection topology, but the message traffic is not oblivious to the data. This sort initially distributes the keys evenly among the processors and at each step picks a pivot, distributes the pivot across the machine, and sends all the keys less than the pivot to one side of the hypercube and the greater keys to the other (this split is always done across the highest dimension of the subcube). This is applied recursively within each subcube. It is very important to pick a pivot that closely balances the two halves otherwise the load on the processors can become extremely imbalanced. Hyperquicksort has been implemented on the NCUBE/10 [39] and the NCUBE/7 [29]. On the NCUBE it was shown to be about twice as fast as bitonic sort, but this was based on randomly generated keys (which would be expected to be a good distribution for the sort). Another variation of quicksort allocates a fair number of processors to each recursive call so that picking bad pivots will not lead to load imbalance [8]. This variation is based on using segmented scans and has been implemented on the CM-2. Because of relatively high communication costs it is not competitive with the sample sort algorithm discussed below.

Another partition based sort is sample sort [16, 19, 30, 34, 41]. Variations of sample sort have been implemented on the CM-2 [9], the Maspar MP-1 [17], the CM-5 [42], and the Ametek/S14 [24] and are the most efficient sorts for most of these machines. Assuming n input keys are to be sorted on a machine with p processors, the sample sort algorithm proceeds in three phases:

1. A set of $p-1$ "splitter" keys are picked that partition the linear order of key values into p "buckets."

2. Based on their values, the keys are sent to the appropriate bucket, where the ith bucket is stored in the ith processor.

3. The keys are sorted within each bucket.

If necessary, a fourth phase can be added to load balance the keys, since the buckets do not typically have exactly equal size.

Sample sort gets its name from the way the $p-1$ splitters are selected in the first phase. From the n input keys, a sample of $ps \leq n$ keys are chosen at random, where s is a parameter called the

oversampling ratio. This sample is sorted, and then the $p-1$ splitters are selected by taking those keys in the sample that have ranks $s, 2s, 3s, \ldots, (p-1)s$. It can be shown that if the samples are picked at random then it is extremely unlikely that any one bucket is more than a small constant larger than the average size bucket [9]: this is true independent of the initial distribution. Some variations of sample sort use splitters that are chosen deterministically [34], but these can have very bad performance with certain key distributions.

The main practical advantage of sample sort is that it greatly reduces the communication required over most of the other sorting algorithms. If there are enough keys per processor (more than p) then the splitters can be broadcast to all the processors without a serious overhead, and the data can be routed to its final destination with a single message. However, because of the need to distribute the splitters and the need to sort the sample, it does not work well when there are a small number of keys per processor. The cost of distributing the p splitters to each processors can be alleviated by running multiple passes [17], but this adds to the communication costs. Another disadvantage of sample sort is that the buckets are not perfectly balanced at the end. This can require extra memory and extra communication to balance the data.

4 Evaluating Sorting Performance

Sorting data is often the dominating cost for any system that makes use of it. It is for this reason that so much has been written and so many algorithms have been developed and analyzed. For any real world application of sorting, however, the order analysis of an algorithm can often be of secondary consideration compared with the constants involved, and it may be that for particular problems certain algorithms may work significantly better than others. We have already seen some of this in the algorithms already mentioned; for uniform data distributions bucket sorts or unbalanced radix algorithms may suffice. For very large keys, counting sorts, such as radix, are unlikely to be optimal and for large amounts of data, partitioning algorithms such as sample sort become optimal. Though generally true, these observations provide no systematic way to evaluate various sorting algorithms in the context of both the sorting job at hand and the computer architecture it is being implemented on. In this section we will formalize some existing terminology for describing sorting and introduce and formalize four dimensions of sorting problems that affect performance.

In choosing a sorting implementation for a particular application there are usually only two main constraints:

1. That the sort be fast.

2. That the data really end up in "sorted" order.

The first constraint is obvious and perhaps the only one usually considered. This is understandable as speed is important and sorting is computationally demanding. It may then seem peculiar that

our second constraint concerns what "sorted" really means. So much has been written about sorting that deciding whether a data set is or is not "sorted" should be well defined. This is not the case. There are many subtle but important variations on sorting. For instance, questions of stability and determinism may be far more important in the choice of a sorting algorithm than overall speed. Even the term "sorted order" is not well defined when considering parallel architectures with distributed memory.

Finding a precise definition of "sorted order" that is independent of machine architecture is difficult. For a serial machine it is generally assumed that "sorted order" requires neighboring elements of the sorted sequence to be allocated to adjoining memory locations. For a distributed memory machine this corresponds to block ordering, where each neighboring element of the sorted sequence is a neighboring element in each processor's memory except for required breaks between processors. However, it also is possible to allocate the sorted keys in a cyclic ordering, where neighboring elements of the sorted sequence are in the memories of neighboring processors. Either allocation pattern may be optimal for different applications but for our purposes we will assume something like block ordering, where neighboring elements in the sorted sequence are "near" each other in memory.

With this definition of sorted order we can state three formal requirements that must be met in order for a data set to be considered sorted:

1. No elements are created or lost.
2. Each successive pair of values in the sorted sequence must pass the comparison test used in sorting the data.
3. Successive pairs of values in the sorted sequence achieve maximal data locality while contained within a single level of the memory hierarchy of the computer system on which the sort is implemented. Preferably, this memory level will be highest in the hierarchy.

4.1 Sorting Descriptors

These three requirements for sorting are rather general and cover almost every implementation of sorting on any architecture. To refine the distinction between sorting implementations, several other descriptors can be used. The user must determine whether these features are necessary as they will affect the overall performance. The following questions should be considered:

1. Is the sort stable?
2. Is the sort deterministic?
3. Is the sort memory efficient?

4. Is the sorted data balanced in memory?
5. Does the cost increase if a rank is performed instead?
6. Is the sort easily extended to include complex keys?

4.1.1 Stability

We have already seen how stability can be important if a counting sort is used as one of the passes for a sort being performed on multiple keys (as in the radix sort). For the radix sort, stability is required for the sort to work at all but it incurs no additional cost. In the cases of partitioned sorts such as sample sort and fixed-topology sorts such as bitonic, stability is accomplished by appending the starting position of the data to the least significant bits of the key. This incurs a cost throughout the algorithm as the element being sorted may be considerably longer than the original key.

4.1.2 Determinism

It may or may not be important to have the sort run in exactly the same amount of time whenever it is called on the same data. The fixed-topology sorts, such as bitonic, are oblivious to the data or its initial allocation in memory and will always take the same amount of time. The counting-based sorts are, however, dependent on the communications network to perform random or irregular permutations. Because of this their running time could vary if the communications network is non-deterministic or if the initial allocation of the data to memory is changed causing different routing patterns which may or may not be more efficient for the given architecture. Partitioning algorithms such as sample sort suffer a similar fate and, in addition, their performance can also vary due to even slight variations in the random sampling of the splitting values. In this case poor sampling could result in an overload for a given bucket which would result in a longer local sort time or even an overflow where the algorithm might have to be restarted with a different random sample.

4.1.3 Memory Efficiency

The maximum amount of additional memory used by a sort in any part of its run determines the maximal number of elements that can be sorted in memory. For external sorting, where the data does not fit within usable memory, it can also affect the run time of the algorithm. This is because the number of passes through the data and the number of random disk accesses performed by most external algorithms is based on the amount of data that can be sorted in memory. Thus it is conceivable that a slower yet more memory efficient sort might be the optimal choice as part of a large external sorting implementation.

4.1.4 Data Balance

For distributed memory parallel computers the third constraint of our basic definition may not be strong enough. Neighboring elements in the sorted sequence may be "near" each other in terms of the memory hierarchy (i.e. they are in successive memory addresses in each processor's memory), but there may be many more sequence elements in one processor's memory than in another's. For some uses of internal sorting this will be satisfactory. In other cases this will produce an imbalance in the amount of processing required of different processor nodes which will result in an overall increased cost. In such cases the data can be easily balanced with an additional enumeration and permutation. In cases such as an external sort where all the data is subsequently permuted and written to disk or in the implementation of a send-with-add permutation where memory collisions are summed, rebalancing the sequence is unnecessary. Counting based sorts and fixed-topology sorts do not result in unbalanced data allocations but partition based sorts such as bucket sort [11] and sample sort usually will.

4.1.5 Rank

Sorts are often used to implement ranks. Normally a rank is preferable to a sort when each key is only a piece of a much larger data element that must be permuted in memory based on the key. Since sorts are often extended in this way it is useful to consider the increase in complexity of the system and performance would be incurred with a rank. A rank can be implemented from a sort by appending a return address to each key and when the keys are in sorted order the enumeration of their positions is returned to this address as the rank. Since stability is often achieved by appending the initial position of the data element to the key, this tag can then also be used as the return address of the rank. For radix and for balanced sample sort this does not even incur the cost of an additional permutation. For fixed-topology sorts such as bitonic, which are not stable, appending the initial position to the key can significantly increase the total cost of the algorithm.

4.1.6 Complex Keys

Though this paper is directed toward current supercomputer applications which typically involve sorting floating point numbers or large integers there are significant applications in sorting of databases where the keys are made up of complex combinations of smaller fields. Any sort that can sort integers can be used to sort floating point values by converting the float into an integer representation that achieves the same sorted order as the float and then translating back to the floating point format. For real world databases such a conversion to a comparable integer format may be difficult. An example might be sorting a payroll database where employees are sorted in descending order by salary and within the groups of employees with the same salaries by last-name and then first name in alphabetic order. With comparison based sorting such as bitonic and some

implementations of sample sort this is made relatively easy by defining a comparison function. It is unlikely that a counting sort such as radix would be universally helpful in this case as the conversion of the complex keys to integers might be difficult and the resulting keys may be rather long.

4.2 Dimensions of Sorting Performance

The implementation or lack of implementation of any of the above descriptors may have an effect on the performance of the sort but in general they are constant multiplicative or additive factors. There are, however, factors that quantitatively affect sorting performance to different degrees depending on their magnitude. Once defined we can look at the performance of a sort along any of these axes and determine how well it will perform for the particular application. We have noted four particular dimensions along which sorting performance should be measured:

1. The number of elements being sorted
2. The size of the key
3. The distribution of key values
4. The initial allocation of data to memory

It would be nice if within this four dimensional space we could give performance figures for various implementations of sorting algorithms. It would also be desirable for there to be a standard measure of performance for sorting. The *MSOPS* (Millions of Sorting Operations per Second) measure has been used in [9] and [37], but unlike the the analogous MFLOPS measure it is highly dependent on many different factors and must be used only within the context of the full description of the sort and the four performance dimensions. For example, consider that for a distributed memory parallel supercomputer a very high MSOPS rating would likely be achieved with a radix sort on a sequence with a small size key requiring a single permutation to sort. Such an MSOPS rating would not reflect the performance for extremely large key sizes where many permutations would be required. Given these caveats, MSOPS is a useful measure of sorting performance.

4.2.1 Number of Keys

The number of keys is often the single most important factor determining the performance of a sorting algorithm. For a given architecture, some algorithms, such as bitonic sort, perform well for a small number of keys. Other algorithms, such as sample sort, pay a high initial cost but become progressive more efficient as the number of keys increases [9].

4.2.2 Key Size

All sorting algorithms are dependent on the size of their key since key size multiplied by the number of keys reflects the total amount of data that must be permuted, and counted or compared. This is true for bitonic sort, sample sort and radix sort. In the case of radix sort, however, the key size also affects the number of permutations that must be performed and the size of the histogram used in counting.

4.2.3 Data Value Distribution

Certain algorithms such as unbalanced radix sort [3] and bucket sort [11] behave poorly when the distribution of data values is non-uniform. In the worst case, where all values are identical, the entire data set will be allocated to and sorted by a single processor. Fixed topology sorts such as bitonic sort are unaffected by data distribution. Other algorithms can exploit non-uniformity in the data. If, for example, the data distribution is sparse (e.g. for 1 million elements there are only 100 different key values for a 64 bit key) it may be advantageous to use a hash table for element counting in the radix sort rather than a histogram. Additionally there may be cases where bits of the key are uniform across all elements and these can be noted and ignored in a radix implementation. To generally categorize the amount of skew of the dataset values Thearling [37] introduced an entropy measure that will be used here also.

4.2.4 Initial Data Allocation

The initial data allocation can also have a significant impact on performance. It is difficult to characterize all possible allocations that might incur performance penalties as this is highly dependent on the machine architecture. However, there are several allocation patterns that are common, such as initially presorted and reverse sorted data. These cases can have significant impact on the performance of the sort. For example, the sample sort can take advantage of presorted data by avoiding most interprocessor communication while fixed topology sorts such as bitonic permute the data between processors equally for any initial data allocation.

5 Important Evaluation Test Cases

Having somewhat formalized the description of sorting problems and the dimensions on which the performance should be measured we now specify some test cases that will exercise sorting implementations along these dimensions. A thorough exploration of each of these dimensions in combination is not possible in a reasonable amount of time. Instead several cases have been chosen that reflect real world problems in both the scientific and commercial community. To keep the test cases to a reasonable number only one parameter is varied at a time. This is not the perfect way to collect

the performance data but it should be sufficient to allow potential users to evaluate the strengths and weaknesses of each sorting implementation. Table 1 summarizes the the base test case and its variations as described below.

5.1 Defining a Base Case

In order to measure the relative changes in performance as each dimension is varied, a base test case is defined consisting of approximately one hundred million elements (2^{27}), 64 bit keys, random values, and randomly allocated distribution. One hundred million elements may seem a high for the base case but it is still one order of magnitude less than the billion element sorts reported in [37] and [3] and should fall squarely in the middle of interesting results in the near future. (It is interesting to note that the size of our base case for internal sorting is eight times the size of what was the standard benchmark for external sorting in 1985 [2]). The base case key length is 64 bits as this corresponds to the common case of sorting double precision floating point numbers.

5.1.1 Variations in Number of Keys

The number of keys is varied from 2^{17} to 2^{34} in multiples of 8. This range is broad enough to exercise any current supercomputer but may have to be expanded in the future.

5.1.2 Variations in Key Size

Though single and double precision floating point numbers (32 and 64 bits) are perhaps of most interest in scientific sorting, smaller keys are also of interest when sorting pointers for example or in some physical applications [5]. With the advent of the commercial use of supercomputers sorting will also be applied to problems where key sizes are very large. When sorting character strings, for example, several textual words, of some five bytes each, will not be uncommon. Thus the key sizes are varied from 8 to 256 bits in logarithmic steps.

5.1.3 Variations in Key Distribution

The key value distribution is varied according to the 6 entropy values presented in [37]. 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?). This section refers 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 [21]. If there are one billion keys, this number is $10^{1166738659}$. 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 [35] of a distribution is defined as $\sum p_i \, |\log p_i|$ where p_i is the probability of symbol i occurring in the distribution. 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 is to simply take a uniform set of keys with 32 bits of entropy and zero out the leading order N bits. This does generate keys with $32 - N$ bits of entropy, but does so effectively by changing the key size and so is undesirable for our purposes. What is desired is a technique for producing keys whose individual bits are between 0 and 1 bit of entropy. There are various techniques for performing this task, and one such method is proposed here.

The basic idea is to combine multiple keys having a uniform distribution into a single key whose distribution is non-uniform. The combination operation to be used is the binary AND. For example, take two 32-bit keys generated using a uniform distribution such that the individual bits as well as 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 the two keys are ANDed 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). Repeating this process using additional uniform keys, the entropies of the key distributions continue to decrease. The difference between successive ANDings is approximately twenty percent of the total for the first five ANDings. The exact percentages (of 1 bit of entropy per binary digit) are as follows: 1 ANDing – 100%, 2 ANDings – 81%, 3 ANDings – 54%, 4 ANDings – 38%, and 5 ANDings – 20%. It takes an infinite amount of additional work to decrease the entropy completely to zero through this procedure. However, zero entropy can be easily obtained by simply setting all of the keys to the same value.

Though this entropy measure captures much of what we would like to notice about non-uniform distributions there are perhaps two common distributions that could be exploited by many algorithms but are not easily constructed in this way. The first is a "sparse distribution" where only 8 bits of the 64 bits of the key are allowed to differ from zero. Thus only 256 different values will actually be obtained in the distribution though there is a possibility of 2^{64}. This distribution can be constructed by generating random 8 bit numbers from 0 through 255 and then inserting seven zeros between each bit to construct a full 64 bit key. This distribution can be exploited by certain radix algorithms that check for variance in each bit of the key before counting or that replace the histogram in the counting step with a hash table.

A further variation which is almost the same as the sparse distribution but which can not be exploited by radix as above would be if 99% of the values were sparse as above but the remaining 1% were random. This distribution can be generated by generating a random number between 0.0 and 1.0 and wherever this value is 0.99 or less generating the key via the above sparse distribution, elsewhere it can be generated from a random number from 0 to 2^{64}.

5.1.4 Variations in Key Allocation

As stated previously, key allocation in memory both before and after the sort is difficult to define since it is architecture dependent. Having decided that "sorted order" corresponds to block order, it is much more difficult to say which layouts of the original data will or will not be difficult for the given architecture to permute. However, there are four cases that are relatively common and should be tried. They include the combination of block layout and cyclic layout for both presorted and reverse sorted data.

5.1.5 Table of Test Cases

Table 1 summarizes the test cases described in the previous subsections. The four parameters to be varied are listed across the top, and the base case is listed in the first row. Below this row there is a 4×4 matrix of blocks where the off-diagonal blocks simply specify base case parameters and the diagonal blocks specify the parameter variation.

6 Conclusion

The ability of a supercomputer to manipulate large amounts of data will determine the future of supercomputers in non-numeric processing, and a standard benchmark in this area is necessary for both algorithmic and technological advances to be gauged. Sorting is a prototypical data movement task which measures a number of important system performance characteristics including communication bandwidth and integer computation performance. It is important in both scientific and

No. of Keys	Key Size	Value Distribution	Allocation Distribution
2^{27}	64 bits	Random	Random
2^{17} 2^{20} 2^{24} 2^{30} 2^{34}	64 bits	Random	Random
2^{27}	8 bits 16 bits 32 bits 128 bits 256 bits	Random	Random
2^{27}	64 bits	Entropy = 0.811 Entropy = 0.544 Entropy = 0.337 Entropy = 0.201 Entropy = 0.0 Sparse 256 Sparse 256/Random	Random
2^{27}	64 bits	Random	Presorted Block Order Presorted Cyclic Order Reverse Sorted Block Order Reverse Sorted Cyclic Order

Table 1: Test cases for supercomputer sorting benchmark.

commercial applications of supercomputers and is a natural choice for a benchmark.

In specifying such a benchmark, variation in the size and distribution of sorted data is necessary to accurately measure how an algorithm/architecture pair performs over a representative range of situations. To this end, a formal set of rules that define a rigorous suite of sorting benchmarks are proposed. These benchmarks requires that large amounts of data be sorted (at least 100 million keys) while varying a number of other parameters (key size, data distribution). These benchmarks can be applied to any of the currently existing supercomputer systems, from single processor vector systems to massively parallel processing systems. With future advances in memory technology, supercomputer systems will be able to process much larger amounts of data. The sorting benchmark will scale with increasing memory capacity to capture the non-numerical performance of future supercomputers.

7 Acknowledgments

The authors would like to thank Steve Heller and Mark Bromley for valuable discussions during the development of this paper, and Eric Barszcz for his many insightful comments on review of the manuscript.

References

[1] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press, Toronto, 1985.

[2] Anon et al. A measure of transaction processing power, In *Datamation*, 1985.

[3] M. Baber. An implementation of the radix sorting algorithm on the Touchstone Delta prototype. In *Proceedings of the Sixth Distributed Memory Computing Conference.*, Portland, Oregon, May 1991.

[4] Bailey, D.H., and Barton, J.T., *The NAS Kernel Benchmark Program*, Technical Report 86711, NASA Ames Research Center, Moffet Field, CA, 1986.

[5] Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L, Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R. S., Simon, H.D., Venkatakrishnan, V., and Weeratunga, S.K., *The NAS Parallel Benchmarks*, The International Journal of Supercomputer Applications, vol. 5, No. 3, pages 63–73, 1991.

[6] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[7] Berry, M., Chen, D., Koss, P., Kuck, D., Lo, S., Pang, Y., Pointer, L., Roloff, R., Sameh, A., Clementi, E., Chin, S., Schneider, D., Fox, G., Messina, P., Walker, D., Hsiung, C., Schwarzmeier, J., Lue, K., Orszag, S., Seidl, F., Johnson, O., Goodrum, R., Martin, J., *The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers*, The International Journal of Supercomputer Applications, vol. 3, No. 3, pages 5–40, 1989.

[8] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, Cambridge, MA, 1990.

[9] 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 Symposium on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head, SC, July 1991.

[10] Carnevali, P., *Timing Results of Some Internal Sorting Algorithms on the IBM-3090*, Parallel Computing, vol. 6, pages 115–117, 1988.

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

[12] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 193–203, May 1990.

[13] L. Dagum. Parallel integer sorting with medium and fine-scale parallelism, *Int J High Speed Computing*, Vol. 5, No. 1, (to appear) 1993.

[14] Dongarra, J., The LINPACK Benchmark: An Explanation, *Supercomputing*, Spring, pp. 10–14, 1988.

[15] Francis, R. S, and Mathieson, I. D., A Benchmark Parallel Sort for Shared Memory Multiprocessors, *IEEE Transactions on Computers*, vol. 37, no. 12, pages 1619–1626, 1988.

[16] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496–507, 1970.

[17] W. L. Hightower, J. F. Prins, J. H. Reif. Implementations of randomized sorting on large parallel machines, In *Proceedings Symposium on Parallel Algorithms and Architectures*, 158–167, July 1992.

[18] C. A. R. Hoare. Quicksort. *Computer J.*, 5(1):10–15, 1962.

[19] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 627–631, November 1983.

[20] S. L. Johnsson. Combining parallel and sequential sorting on a Boolean n-cube In *Proceedings of the International Conference on Parallel Processing*, pages 444–448, August, 1984.

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

[22] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, San Mateo, CA, 1992.

[23] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34(4):344–354, April 1985.

[24] P. P. Li. Parallel sorting on Ametek/S14. Technical report, Ametek Computer Research Division, Arcadia, CA, September 1986.

[25] G. Manzini. Radix sort on the hypercube. Information Processing Letters, 38(2):77–81, April 1991.

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

[27] Moscinski, J., Rycerz, Z. A., and Jacobs, P. W. M., *Timing Results of Some Internal Sorting Algorithms on the ETA 10-P*, Parallel Computing, vol. 11, pages 117–119, 1989.

[28] J. F. Prins. Efficient Bitonic sorting of large arrays on the MasPar MP-1 Proceedings of the 3rd Symposium on Frontiers of Massively Parallel Computation, pages 158–167, October, 1990.

[29] M. J. Quinn. Analysis and benchmarking of two parallel sorting algorithms: hyperquicksort and quickmerge. *BIT*, 29(2):239–250, 1989.

[30] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[31] D. Richards. Parallel sorting—a bibliography. ACM SIGACT News, 28–48, 1986.

[32] W. Rönsch and H. Strauss. Timing results of some internal sorting algorithms on vector computers. *Parallel Computing*, 4, 49–61, 1987.

[33] R. Sedgewick. Implementing quicksort programs. Communications of the ACM, 21(10):847–857,1978.

[34] S. R. Seidel and W. L. George. Binsorting on hypercubes with d-port communication. In *Proceedings of the Third Conference on Hypercube Concurrent Computers*, pages 1455–1461, January 1988.

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

[36] T. M. Stricker. Supporting the hypercube programming model on mesh architectures (A fast sorter for iWarp tori). In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 148–157, July 1992.

[37] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Proceedings Supercomputing '92*, pages 14–19, November 1992.

[38] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer, Communications of the ACM, 20(4):263–271, 1977.

[39] B. A. Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. In M. T. Heath, editor, *Hypercube Multiprocessors 1987 (Proceedings of the Second Conference on Hypercube Multiprocessors)*, pages 292–299, Philadelphia, PA, 1987. SIAM.

[40] K. White, H. Sheng. An Efficient Multiprocessor Column Sort Algorithm on the Connection Machine CM-5. Unpublished manuscript, Department of Electrical and Computer Sciences, University of California, Berkeley.

[41] Y. Won and S. Sahni. A balanced bin sort for hypercube multicomputers. *Journal of Supercomputing*, 2:435–448, 1988.

[42] M. Zagha. Sorting algorithms for the Connection Machine CM-5, presentation at Thinking Machines Corporation, September 17, 1992.

[43] M. Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, November 1991.