A COMPARISON BASED PARALLEL SORTING ALGORITHM*

Laxmikant V. Kalé
Department of Computer Science
University of Illinois
Urbana, IL 61801
E-mail: kale@cs.uiuc.edu

Sanjeev Krishnan
Department of Computer Science
University of Illinois
Urbana, IL 61801
E-mail: sanjeev@cs.uiuc.edu

Abstract

We present a fast comparison based parallel sorting algorithm that can handle arbitrary key types. Data movement is the major portion of sorting time for most algorithms in the literature. Our algorithm is parameterized so that it can be tuned to control data movement time, especially for large data sets. Parallel histograms are used to partition the key set exactly. The algorithm is architecture independent, and has been implemented in the CHARM portable parallel programming system, allowing it to be efficiently run on virtually any MIMD computer. Performance results for sorting different data sets are presented.

1 Introduction

Sorting is one of the most basic algorithms in computer science. As a parallel application, sorting it is challenging because of the extent of communication it requires. Essentially, almost all data items must move from the processor they were originally on to some other processor. Moreover, in a network of processors, the average number of hops travelled by each data item is of the order of the network diameter.

The input to a sorting algorithm is a collection of records. Each record has a designated *key* field and possibly multiple data fields. Applications of sorting in different contexts may involve a variety of data types as keys. The keys may be integers, floating point number, long strings of characters, or records of arbitrary structure. Comparison based sorting methods only assume the existence of an operation which compares two keys and determines if one is smaller, larger or equal to the other, in some metric. Thus they are more general than methods such as radix sort which depend on knowledge of internal structure of the keys and its relationship with the underlying ordering.

In this paper we present a new comparison based parallel sorting algorithm, analyze its complexity, present performance results, and compare it with previous work.

2 The algorithm

We assume that there are initially n keys distributed among p processors such that each processor has approximately n/p keys[1]. At the end of sorting, the data must be approximately equally distributed among the processors, and for $i = 1$ to $n-1$, all keys on processor $i-1$ should be less than any key on processor i. In other words, processor 0 must have the smallest n/p keys, processor 1 must have the next n/p keys, and so on. The data within each processor must be in sorted order.

2.1 Overview of algorithm

The basic structure of the algorithm is similar to sample sort [4, 5, 6], load balanced sort [1], hyperquicksort [10] and binsort [11], in that in each phase, it finds $k-1$ "splitter" keys that partition the linear order of keys into k equal partitions. These keys are found by an initial local sort on each processor followed by repeated iterations of global histogramming (Section 2.2). Each of the k partitions is then sent to the appropriate set of p/k processors such that the i^{th} processor partition gets the i^{th} data partition (Section 2.3). The next recursive phase of the algorithm can then run independently in all k processor partitions. Figure 1 gives a high level view of the algorithm.

k is the parameter that controls data movement. When k is equal to p, there is only one phase, and every key moves exactly once. At the other extreme, when k is 2, the algorithm essentially finds the median of the key set, then each processor sends all keys less than the median to the first $p/2$ processors and all keys greater than the median to the last $p/2$ processors, thus resulting in $log_k(p)$ phases of data movement.

2.2 Data Partitioning with Histograms

The object of this step is to find the $k-1$ splitter keys, (partition boundaries) defined as the keys having ranks n/k, $2*n/k$ ... $(k-1)*n/k$ in the global order of keys. This step of the algorithm consists of a series of iterations consisting of upward passes (called *reductions*) and broadcasts along a logarithmic spanning tree. Starting with an initial set of splitter keys, each iteration refines the splitter keys till all partitions (as defined by the splitter keys) have approximately equal number of keys.

At the beginning of this step, each processor has a sorted key set. The first set of splitter keys can be found in var-

*This research was supported in part by the National Science Foundation grants CCR-90-07195 and CCR-91-06608.

[1] If the initial load distribution is unbalanced, the performance of the algorithm may degrade in proportion to the degree of imbalance. If so, we can use a load distribution phase similar to that in Section 2.3.

```
Perform Local Sort on each processor.
for ( phase = 1 to log_k p )
    do /* This is the Histogramming step */
        Generate histogram probes and broadcast them.
        On each processor find key counts for probes.
        Send counts up spanning tree to root.
        At root processor, use new set of counts to refine
            current best values of partition boundaries.
    while ( key counts for each partition are unequal )
    At root, generate k quintuples per subtree (section 2.3).
    Send quintuples down spanning tree. At each internal
    node, split each quintuple among subtrees.
    On each processor, use k quintuples to find what data
    to send to which processors.
    On each processor, send keys to other processors,
    then merge keys received from other processors.
    Reconfigure tree into k separate spanning trees,
    one for each partition.
endfor
```

Figure 1: High level description of algorithm. n is the total number of keys, p is the number of processors, k is the number of partitions

ious ways (see Section 6). In our current implementation we find the keys that equally divide the key set of the processor at the spanning tree root. If the root processor's distribution is representative of the global distribution, the splitter keys found by considering only the root would be reasonably good.

The number of probes (histogram boundaries) m may be more than the number of partition boundaries (number of splitter keys) $(k-1)$. Making m larger helps us refine the splitter keys faster, but increases the size of messages going up and down the spanning tree. We use $m = 3 * (k-1)$ as a heuristic, with $k-1$ of the m values being the best guesses for the splitter keys, another $k-1$ being slightly lesser than each of the first $k-1$, and the last $k-1$ being slightly greater than each of the first $k-1$.

The m probes are arranged in sorted order in a single message and broadcast to all processors using the spanning tree. Each processor then counts the number of keys less than or equal to each of the m probes. (A simple binary search is used). The array of m counts is sent up to the root by a reduction pass, with combining at internal nodes of the spanning tree, so that all message sizes are m. At internal nodes the counts received from subtrees are stored for use later during the data movement step.

The root maintains the current best known lower and higher values for each of the splitter keys. These values are updated using the new set of probes and their corresponding counts as follows : if any of the counts that came up the tree is nearer to the desired partition boundary (e.g. $n/2$ for the median) than the current best known count, then that count is replaced with this new count and its corresponding probe value. If the partition counts as specified by the current splitter keys are not equal (within some user specified tolerance factor), the root calculates a new set of splitter keys. Each of the $k-1$ splitter key values is found by proportional linear interpolation[2] between the current best lower and higher values and their corresponding current best lower and higher counts, using the count for the desired partition boundary.

The root then starts the next histogram refinement iteration. If the partition counts are equal, the splitter keys have been successfully found, and the data movement step can be started.

2.3 Data Movement

Once the root has determined the splitter keys that partition the key set, we can efficiently move keys to their destination partitions as described below.

We initially have n/k keys in each partition distributed (possibly unequally) among p processors, which have to be transferred to a set of p/k processors such that each of the p/k processors ends up with approximately the same number of keys. Thus each processor must know whom to send to, as well as how many keys to send. We also want to ensure that each key moves exactly once per phase, hence it is not acceptable for a representative processor in each partition to collect all the keys for that partition and then distribute it among the processors of that partition.

The root knows at the beginning how many keys each partition has, but does not know which processors have those keys. However, this information is available at the internal nodes of the spanning tree. Hence in one downward pass, it is possible to tell each processor how many keys to send to which processors.

The entities being passed down the spanning tree are *constant sized* (5 integers) *quintuples* of the form *(startproc, startdata, middledata, endproc, enddata)* which indicates that processor *startproc* is supposed to receive *startdata* keys, processor *endproc* receives *enddata* keys, and the processors in between receive *middledata* keys each. There is a quintuple for each partition, hence the messages consist of k quintuples.

When a processor i at an internal node of the spanning tree receives a quintuple for a particular partition x, it means that all the keys in the subtree rooted at i belonging to partition x must be sent to the processors specified in the quintuple. Processor i has the latest histogram counts for its subtrees (which came from each of its subtrees during the last histogram refinement), hence it has information about how many keys each subtree has in each partition. This information is used to derive the quintuples for the subtrees from the quintuple that came from the parent.

[2] Interpolation is not strictly a comparison operation. However, an interpolation function can be provided for many cases where a comparison function is available. In Section 3 we prove that this step can be accomplished solely with a comparison function

At the root the global quintuple for the first partition would be $(0, n/p, n/p, p/k-1, n/p)$. This is divided among its subtrees depending on how many keys each subtree has in that partition. Thus each internal node of the spanning tree receives a quintuple per partition from its parent and divides it among its children.

Finally, each processor has a quintuple per partition, indicating how many keys it has to send to which other processors for that partition. Now all sends and receives of keys for all partitions occur in parallel. Each processor sends the part of its local key set belonging to a partition to one or more processors in the set of p/k processors corresponding to the destination of that partition.

Note that in both data partitioning and data movement steps, the upward and downward passes along the spanning tree involve constant sized messages (i.e. the size is independent of the number of keys or number of processors) at all heights in the spanning tree, because of combining at internal nodes.

3 Complexity Analysis

We prove a bound on the number of histogramming iterations required, assuming no interpolation function is available. This bound is independent of the number of partitions because histogramming for all partitions is done together.

We consider the case where the number of partitions k is 2 , which means that we want to find the median of the key set. We maintain an upper and a lower bound on the value of the median. One of these bounds is refined in each histogramming iteration. We first prove that each refinement iteration decreases the number of keys between the bounds by a constant multiplicative factor c. Let n_i be the number of keys between the bounds at the i^{th} iteration. Let m_i be the value to be guessed for the median at the i^{th} iteration. Let j be the processor that has the most number of keys between the bounds. Processor j must have at least $n_i^j = \frac{n_i}{p}$ keys between the bounds. We choose m_i as the median of these n_i^j keys. Now after we get a global histogram, m_i may turn out to be either lesser or greater than the median we seek, so m_i becomes the new lower or upper bound, respectively. In either case, we have decreased the keys between the bounds by at least $\frac{n_i}{2*p}$. Thus in the worst case,

$n_{i+1} = n_i - \frac{n_i}{2*p}$

$n_{i+1} = n_i/c$ where $c > 1$.

Using this result we can prove a bound on the number of refinements. Let $n_0 = n$ be the initial number of keys, then

$n_i = n_0/c^i$.

If t is the total number of iterations required, then at the end of t iterations, the number of keys between the upper and lower bounds is 1, hence $n_t = 1$. Thus

$1 = n_0/c^t$, hence

$\mathbf{t = O(\log(n))}$.

It must be emphasized, however, that in practice, the number of histogram iterations required is much smaller than this worst case (depending on the data distribution, see Table 2). This is because we have an interpolation function, and we have more than one probe point, which allows us to converge towards the median much faster.

Using the above result, the total time taken by the algorithm is :

local sort time + $\lceil log_k(p) \rceil \{O(log(n))$(time per histogram) + data movement and merging time $\}$

4 Implementation and results

We have implemented our algorithm in the CHARM portable parallel programming system [7]. CHARM supports C with a few extensions for creation of tasks and message passing. CHARM has a message driven model of execution, allowing overlap of computation and communication. Our implementation can run without change on nonshared as well as shared memory machines.

4.1 Basic results

Table 1 gives performance results for the nCUBE/2 and Intel iPSC/860 systems. The data set consists of 2^{23} integers formed by averaging four sets of random numbers as in the NAS Integer Sort Benchmark [3]. All random numbers for these measurements were generated using the C library function `lrand48`. Measurements for this table were taken with $k = 8$ and the tolerance factor as 1% (which means that the *final* counts of keys per processor can differ by only 1% of the total number of keys). The timings in all tables do not include startup, data generation and correctness checking times. From the table we can see good speedups as the number of processors increases. The results for the iPSC/860 compare well with other results reported in the literature [3, 2]. Considering the fact that parallel sorting is inherently a communication intensive application, these results demonstrate that our algorithm successfully reduces communication.

Table 1: Histogram Sort Basic Timings on the nCUBE/2 and iPSC/860. The keys are integers obtained by averaging 4 sets of random integers.

Number of Processors	Number of Keys	nCUBE/2 (s)	iPSC/860 (s)
64	2^{23}	12.30	3.87
128	2^{23}	6.87	2.66
128	2^{24}	-	5.04
256	2^{23}	3.93	-
512	2^{23}	2.46	-
1024	2^{23}	2.00	-
1024	2^{26}	9.14	-

4.2 Effect of data distribution

The performance of our algorithm depends, to some extent on the probability distribution of data in the space of possible key values. In general, uniform, random distributions are easier to sort as compared to non-uniform distri-

butions having significant amounts of data concentrated in small value ranges.

Entropy [9, 8] has been suggested as a metric of distribution. Informally, the entropy of a key set corresponds to the number of "unique" bits in the key. However, entropy suffers from the drawback that low entropies (which mean that some bits are effectively unused) need not necessarily mean non-uniform distributions, and conversely, uniform distributions need not have high entropies. Consider a distribution consisting of equal numbers of all integers whose *least* significant 16 bits are 0. This distribution is uniform, but has entropy of 16 bits. Consider another distribution consisting of all integers whose *most* significant 16 bits are 0. This also has an entropy of 16 bits, but is highly nonuniform, in that all the data is concentrated in $1/2^{16}th$ of the data space. For most algorithms, this non-uniformity makes a difference in performance, still entropy usually will not distinguish between these distributions. Thus data distribution cannot be characterized by a single metric such as entropy.

Table 2 gives performance results for different distributions. These timings were taken on the nCUBE/2 with 256 processors, with number of partitions (k) as 8, tolerance factor 1% and a data set of 2^{23} integers. Distribution D1 is a uniform random distribution. Distribution D2 was generated by averaging four sets of random numbers. Distribution D3 has entropy 25.95 and was generated by performing a bitwise AND operation on two sets of random integers [9]. Distribution D4 has only 4 distinct values for the most significant 16 bits, while the least significant 16 bits have uniformly distributed random values. Distribution D4 has entropy 10.78 and was generated by doing an AND on four sets of random integers.

Although the table shows widely differing distributions, (from uniform, random to highly non-uniform), the histogramming time increases only slightly, from about 4 % to 10 % of the total execution time. This demonstrates that histogramming is an efficient method for obtaining the pattern of data distribution. This is because algorithm makes effective use of short, constant sized messages moving along the spanning tree.

Table 2: Timings as a function of distribution for sorting 2^{23} integers on the nCUBE/2 with 256 processors. There are 3 phases, because $k = 8$.

Distribution	Histogram iterations per phase	Histogramming Time (%)	Total Time (s)
D1	3+3+2	4.4	4.00
D2	4+3+2	5.0	3.93
D3	11+8+3	10.1	4.23
D4	12+5+2	10.4	4.28
D5	15+3+1	8.2	5.20

Table 3 gives a breakup of the times among the various steps. These timings were taken on the nCUBE/2, with a data distribution as for Table 1. It can be seen that the time spent in communication, which is included completely in the sum of the histogramming and data movement times, (15 % on the average for the nCUBE), is not the major portion of the total execution time. (For the iPSC/860 about 45 % of the time was spent in communication). Moreover, data movement time decreases as a fraction of total time when k increases. In general, the optimum value of k depends on the number of processors. For small k, each data movement step takes less time because there are fewer messages, but there more phases. The opposite happens for large k. For 1024 processors, $k = 8$ was observed to be optimal.

Table 3: Breakup of timings for sorting 2^{23} integers on the nCUBE/2. Hist and Move are the times for histogramming and data movement, respectively.

Processors, Partitions, Phases	Local Sort (%)	Hist (%)	Move (%)	Total Time (s)
128, 4, 4	43.8	2.2	12.6	7.27
128, 8, 3	46.4	2.1	10.4	6.87
128, 16, 2	49.5	2.7	8.1	6.44
128, 128, 1	46.8	7.9	7.9	6.81

5 Previous work

The fast parallel sorting algorithms reported in the literature have been mostly non comparison-based ones, such as those based on radix-sort [2, 9]. Even with lexicographically-ordered data (such as names), radix sort based methods are inefficient or impossible to use, if the length of the keys is large and variable (and possibly unbounded). In addition, for non lexicographically-ordered data, such methods cannot be used at all.

For example, consider a set of customer records maintained by a consumer-service company. They wish to sort such records in decreasing order of importance/attention the company wishes to accord to customers (for a sophisticated mass mailing, say). Given two customer records, one can use a subroutine based on heuristics that decides which customer is worthy of more attention. However, the heuristic knowledge embodied in such a routine cannot always be extracted and quantified to yield a single numeric metric of importance. In such a situation, a comparison based sort can be used, while a radix sort cannot be used.

Moreover, for large data sets, most of the time for radix sort is taken up by data movement. For 32 bit keys, it is infeasible to use a 32 bit radix (that would involve 2^{32} buckets), hence at least two data moves involving all processors are required. In our algorithm, we can set k equal to p, so that only one move of data is required. Even if we set $k = \sqrt{p}$ so that two moves are required, the second move involves only $\sqrt{p}$ processors, hence will have less cost.

Load balanced sort [1] has the same high level steps as our algorithm. However, our data partitioning step avoids the transpose operation, and moreover, can use

more probes than partitions, for faster convergence. Load balanced sort is a special case of our algorithm, with $k = p$, hence our algorithm has more flexibility in face of differing communication parameters. Finally, our algorithm does not depend on the topology of the underlying machine.

Our algorithm has the same high level steps as sample sort [4, 5, 6], with some important differences :

- Data Partitioning in our algorithm is exact; the time (number of histogram iterations) taken depends on the input distribution : for uniform distributions, one or two iterations are enough, hence an almost exact partition can be found quickly. For non-uniform cases the time taken depends on the tolerance (deviation from exact partitioning) specified by the user. Sample sort does not optimize the uniform case.
- Sample sort is not scalable, because the size of the messages that need to be communicated is $samplesize * O(p)$, which is usually $64 * p$ *keys* that could be large [4]. In our algorithm, all messages going up and down the spanning tree have size $O(k)$, which in the worst case is $5 * k$ *integers*, and the height of the spanning tree is $O(log(p))$. No keys are moved except in the actual data movement step.

The other salient features of our algorithm that differentiate it from most, if not all of the algorithms reported in the literature are :
- The number of partitions may be less than the number of processors, allowing the algorithm to run in more than one recursive phase, if that is faster.
- A novel method using histograms is used to find the partitions.
- All messages except those actually carrying keys are constant sized (independent of number of processors). Thus reductions take very little time.
- Key movement is reduced by requiring keys to move exactly once per phase.
- The algorithm does not depend on any particular architecture.

6 Discussion and Conclusions

We have described a comparison-based parallel sorting algorithm, and demonstrated its performance on a large parallel machine. The algorithm can be tuned to widely varying combinations of number of processors, data-sizes, and machine communication parameters by varying k, the number of partitions in each phase (and thus, the number of phases).

Further refinements to the sorting algorithm that we believe will improve its performance include
- A possible trade off involves doing away with the initial sorting phase, at the cost of spending more time on producing the histogram for a given probe with unsorted keys.
- Since the optimal value of k depends on the number of processors, we can allow k to dynamically vary from phase to phase, depending on the number of processors in a phase.
- The initial probe keys for the first histogramming iteration in each phase can be generated by more sophisticated methods. For example, each processor can find its local set of splitter keys, which can be combined heuristically in a reduction, so that a more accurate picture of data distribution is obtained.
- The message driven nature of CHARM allows us to overlap communication and computation effectively. Thus the histogram phase itself can be pipelined by segmenting the set of probe keys and the corresponding histogram into separate pieces that can be broadcast and reduced concurrently with each other. The data movement step is currently overlapped with the merging step, and we could also overlap the histogramming and data movement steps.

Finally, we expect the algorithm to perform very well on MIMD computers with faster processors, because the time for local computations, which is the major fraction of the total time, will be reduced.

References

[1] B. Abali, F. Ozguner, and A. Bataineh. Load balanced sort on hypercube multiprocessors. In *Proc. Fifth Distributed Memory Computing Conference*, Apr. 1990.

[2] M. Baber. An implementation of the radix sorting algorithm on the Touchstone Delta Prototype. In *Proc. Sixth Distributed Memory Computing Conference*, May 1991.

[3] D. Bailey, E. Barszcz, L. Dagum, and H. Simon. NAS parallel benchmark results. In *Proc. Supercomputing*, Nov. 1992.

[4] G. Blelloch et al. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. Symposium on Parallel Algorithms and Architectures*, July 1991.

[5] W. Fraser and A. McKellar. Samplesort : A sampling approach to minimal storage tree sorting. *Journal of the Association for Computing Machinery*, 17(3), July 1970.

[6] J. Huang and Y. Chow. Parallel sorting and data partitioning by sampling. In *Proc. Seventh International Computer Software and Applications Conference*, Nov. 1983.

[7] L. Kale. The Chare Kernel parallel programming language and system. In *Proc. International Conference on Parallel Processing*, Aug. 1990.

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

[9] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Proc. Supercomputing*, Nov. 1992.

[10] B. Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. In *Proc. Second Conference on Hypercube Multiprocessors*, Sept. 1986.

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