A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation

Abstract

We introduce a new deterministic parallel sorting algorithm based on the regular sampling approach. The algorithm uses only two rounds of regular all-to-all personalized communication in a scheme that yields very good load balancing with virtually no overhead. Moreover, unlike previous variations, our algorithm efficiently handles the presence of duplicate values without the overhead of tagging each element with a unique identifier. This algorithm was implemented in Split-C and run on a variety of platforms, including the Thinking Machines CM-5, the IBM SP-2-WN, and the Cray Research T3D. We ran our code using widely different benchmarks to examine the dependence of our algorithm on the input distribution. Our experimental results illustrate the efficiency and scalability of our algorithm across different platforms. In fact, the performance compares closely to that of our random sample sort algorithm, which seems to outperform all similar algorithms known to the authors on these platforms. Together, their performance is nearly invariant over the set of input distributions, unlike previous efficient algorithms. However, unlike our randomized sorting algorithm, the performance and memory requirements of our regular sorting algorithm can be deterministically guaranteed.

1 Introduction

We present a novel variation on the approach of sorting by regular sampling which leads to a new deterministic sorting algorithm that achieves optimal computational speedup with very little communication [15]. Our algorithm exchanges the single step of irregular communication used by previous implementations for two steps of regular communication. In return, our algorithm reduces the problem of poor load balancing because it is able to sustain a high sampling rate at substantially less cost. In addition, our algorithm efficiently accommodates the presence of duplicates without the overhead of tagging each element. And our algorithm achieves predictable, regular communication requirements which are essentially invariant with respect to the input distribution. Utilizing regular communication has become more important with the advent of message passing standards, such as MPI [17], which seek to guarantee the availability of very efficient (often machine specific) implementations of certain basic collective communication routines.

Our algorithm was implemented in a high-level language and run on a variety of platforms, including the Thinking Machines CM-5, the IBM SP-2, and the Cray Research T3D. We ran our code using a variety of benchmarks that we identified to examine the dependence of our algorithm on the input distribution. Our experimental results are consistent with the theoretical analysis and illustrate the efficiency and scalability of our algorithm across different platforms. In fact, the performance compares closely to that of our random sample sort

algorithm, which seems to outperform all similar algorithms known to the authors on these platforms. Together, their performance is nearly indifferent to the set of input distributions, unlike previous efficient algorithms. However, unlike our randomized sorting algorithm, the performance and memory requirements of our regular sorting algorithm can be guaranteed with probability one.

The high-level language used in our studies is Split-C [10], an extension of C for distributed memory machines. The algorithm makes use of MPI-like communication primitives but does not make any assumptions as to how these primitives are actually implemented. The basic data transport is a read or write operation. The remote read and write typically have both blocking and nonblocking versions. Also, when reading or writing more than a single element, bulk data transports are provided with corresponding bulk read and bulk write primitives. Our collective communication primitives, described in detail in [6], are similar to those of the MPI [17], the IBM POWERparallel [7], and the Cray MPP systems [9] and, for example, include the following: transpose, bcast, gather, and scatter. Brief descriptions of these are as follows. The transpose primitive is an all-to-all personalized communication in which each processor has to send a unique block of data to every processor, and all the blocks are of the same size. The bcast primitive is used to copy a block of data from a single source to all the other processors. The primitives gather and scatter are companion primitives. Scatter divides a single array residing on a processor into equal-sized blocks, each of

which is distributed to a unique processor, and gather coalesces these blocks back into a single array at a particular processor. See [3, 6, 4, 5] for algorithmic details, performance analyses, and empirical results for these communication primitives.

The organization of this paper is as follows. Section 2 presents our computation model for analyzing parallel algorithms. Section 3 describes in detail our improved sample sort algorithm. Finally, Section 4 describes our data sets and the experimental performance of our sorting algorithm.

2 The Parallel Computation Model

We use a simple model to analyze the performance of our parallel algorithms. Our model is based on the fact that current hardware platforms can be viewed as a collection of powerful processors connected by a communication network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. We view a parallel algorithm as a sequence of local computations interleaved with communication steps, where we allow computation and communication to overlap. We account for communication costs as follows.

Assuming no congestion, the transfer of a block consisting of m contiguous words between two processors takes (o + oem) time, where o is the latency of the network and oe is the time per word at which a processor can inject or receive data from the network. Note that the

bandwidth per processor is inversely proportional to oe. We assume that the bisection bandwidth is sufficiently high to support block permutation routing amongst the p processors at the rate of 1oe . In particular, for any subset of q processors, a block permutation amongst the q processors takes (o + oem) time, where m is the size of the largest block.

Using this cost model, we can evaluate the communication time Tcomm(n; p) of an algorithm as a function of the input size n, the number of processors p , and the parameters o and oe. The coefficient of o gives the total number of times collective communication primitives are used, and the coefficient of oe gives the maximum total amount of data exchanged between a processor and the remaining processors.

This communication model is close to a number of similar models (e.g. [11, 19, 1]) that have recently appeared in the literature and seems to be well-suited for designing parallel algorithms on current high performance platforms.

We define the computation time Tcomp as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance Tcomp+Tcomm involves a tradeoff between Tcomp and Tcomm. In many cases, it is possible to minimize both Tcomp and Tcomm simultaneously, and sorting is such a case.

3 A New Sorting Algorithm by Regular Sampling

Consider the problem of sorting n elements equally distributed amongst p processors, where we assume without loss of generality that p divides n evenly. The idea behind sorting by regular sampling is to find a set of p ? 1 splitters to partition the n input elements into p groups indexed from 1 up to p such that every element in the ith group is less than or equal to each of the elements in the (i + 1)th

group, for (1 <= i <= p ? 1). Then the task of sorting each of the p groups can be turned over to the correspondingly indexed processor, after which the n elements will be arranged in sorted order. The efficiency of this algorithm obviously depends on how well we divide the input, and this in turn depends on how evenly we choose the splitters. One way to choose the splitters is by regularly sampling the sorted input elements at each processor - hence the name Sorting by Regular Sampling. A previous version of regular sample sort [18, 16], known as Parallel Sorting by Regular Sampling (PSRS) , first sorts the np elements at each processor and then selects every ? np2 ?th

element as a sample. These samples are then routed to a single processor, where they are sorted and every pth sample is selected as a splitter. Each processor then uses these splitters to partition the sorted input values and then routes the resulting subsequences to the appropriate destinations, after which local merging of these subsequences is done to complete the sorting process. The first difficulty with this approach is the load balance. There exist inputs for which at least one processor will be left with as many as

?
2np ? np2 ? p + 1
?
elements at the completion of sorting. This could be reduced by choosing more samples, but this would also increase the overhead. And no matter how many samples are chosen, previous studies have shown that the load balance would still deteriorate linearly with the number of duplicates [16]. One could, of course, tag each item with a unique value, but this would also double the cost of both memory access and interprocessor communication. The other difficulty is that no matter how the routing is scheduled, there exist inputs that give rise to large variations in the number of elements destined for different processors, and this in turn results in an inefficient use of the communication bandwidth. Moreover, such an irregular communication scheme cannot take advantage of the regular communication primitives proposed under the MPI standard [17]. In our algorithm, which is parameterized by a sampling ratio s
?
p <= s <= np2
?
, we guarantee that, at the completion of sorting, each processor will have at most
?np + ns ? p
?
elements, while incurring no overhead in gathering the set of samples used to identify the splitters. This bound holds regardless of the number of duplicate elements present in the input. Moreover, we are able to replace the irregular routing with exactly two calls to our transpose primitive. The pseudocode for our algorithm is as follows:

ffl Step (1): Each processor Pi (1 <= i <= p) sorts each of its np input values using an appropriate sequential sorting algorithm. For integers we use the radix sort algorithm, whereas for floating

point numbers we use the merge sort algorithm. The sorted data is then \dealt" into p bins so that the kth item in the sorted order is placed into the

ffl Step (2): Each processor Pi routes the contents of bin j to processor Pj , for (1 <= i; j <= p), which is equivalent to performing a transpose operation with block size np2 .

ffl Step (3): From each of the p sorted subsequences received in Step (2), processor Pp selects each
?

ffl Step (4): Processor Pp merges the p sorted subsequences of samples and then selects each (ks)th

sample as Splitter[k], for (1 <= k <= p? 1). By default, the pth splitter is the largest value allowed by the data type used. Additionally, binary search is used to compute for the set of samples

Sk with indices (ks ? s + 1) through (ks) the number of samples Est[k] which share the same value as Splitter[k].

ffl Step (5): Processor Pp broadcasts the Splitter and Est arrays to the other p ? 1 processors.

ffl Step (6): Each processor Pi uses binary search to define for each of the p sorted sequences received in Step (2) and each of the p splitters a subsequence T(j;k). The set of p subsequences fT(j;1); T(j;2); :::; T(j;p)g associated with Splitter[j] all contain values which are greater than or equal to Splitter[j ? 1] and less then or equal to Splitter[j], and collectively include at most ?
Est[j] ? np2 s
?
elements with the same value as Splitter[j].

ffl Step (7): Each processor Pi routes the p subsequences associated with Splitter[j] to processor Pj , for (1 <= i; j <= p). Since no two processors will exchange more than
? np2 + nsp
?
elements, this
is equivalent to performing a transpose operation with block size
? np2 + nsp
?
.

ffl Step (8): Each processor Pi \unshuffles" all those subsequences sharing a common origin in Step (2).

ffl Step (9): Each processor Pi merges the p consolidated subsequences to produce the ith column of the sorted array.

Before establishing the complexity of this algorithm, we need to establish the following theorem.

Theorem 1: The number of elements sent by processor Pi to processor Pj in Step (7) is at most Consequently, at the completion of the algorithm, no processor holds more than

.

Proof: Let Sj be the set of samples from the sorted array of samples in Step (4) with indices (js?s+1) through (js), inclusively. Let S(j;i) be the subset of samples in Sj originating from processor Pi, and let c(j;i) be the cardinality of S(j;i). Let Vj = Splitter[j], let c(j;i;1) be the number of samples in S(j;i) with

value less than Vj, and let c(j;i;2) = (c(j;i) ?c(j;i;1)) be the number of samples in S(j;i) with value Vj . Let R(j;i) = f[S(t;i) : 1 <= t < (j ? 1)g, let b(j;i) = Pj?1
t=0 c(t;i) be the cardinality of R(j;i), let b(j;i;1) be the number of samples in R(j;i) with value less than Vj , and let b(j;i;2) = (b(j;i) ? b(j;i;1)) be the number of samples in R(j;i) with value Vj

. Obviously, b(j;i;2) will only be nonzero if Splitter[j ? 1] = Vj .
Finally, for simplicity of discussion but without loss of generality, we assume that n is a constant multiple of p2s.

Clearly, each sample can be mapped in a one-to-one fashion to the sorted input generated during Step (1) (but before being distributed amongst the bins). For example, the first sample in S(j;i) maps to the

?th element in the sorted input at processor Pi, the second sample in

?th element in the sorted input at processor Pi, and so forth up to the

cth(j;i) element which maps to the

?th element in the sorted input at processor Pi.
Hence, it follows that L(j;i) elements in the sorted input of Step (1) at processor Pi will be less than Vj , where

. It is also true that
at least M(j;i) elements in the sorted input of Step (1) will be less than or equal to Vj, where M(j;i) =
?

The shuffling of Step (1) together with the transpose of Step (2) maps the tth element at processor Pi into the
?j t?1p
k
+ 1
?th position of the ith subarray at processor P(((t?1) mod p)+1), a subarray which we will denote as SA((((t?1) mod p)+1);i). Now, L(j;r;i) elements in SA(r;i) will be less than Vj and will unequivocally route to processors P1 through Pj , where:

Furthermore, at least M(j;r;i) elements in SA(r;i) will be less than or equal to Vj, where M(j;r;i) = (b(j;i) + c(j;i)) np2 s). This means that the p subarrays at processor Pr collectively have at least

elements which are greater than or equal to Vj . Furthermore,

is the number of elements equal to Vj which the algorithm in Step (6) will
seek to route to processor Pj and

is the number of elements equal to Vj which the
algorithm in Step (6) will seek to route to processors P1 through P(j?1). From this it follows that the algorithm will always be able to route a minimum of Min(j;r) = Ppt=1 M(j;r;t) = j np2 elements to processors P1 through Pj . On the other hand, the maximum number of elements that will be routed by this algorithm to these processors is:

Hence, the maximum number of elements send by processor Pr to processor Pj is:

and Theorem 1 follows.

With the results of Theorem 1, the analysis of our algorithm for sorting by regular sampling is as follows. Steps (3), (4), (6), (8), and (9) require O(sp), O(sp log p), O ?p2 log p?, O ?np + ns + p2 ? p

time, respectively. The cost of sequential sorting in Step (1) depends on
the data type - sorting integers using radix sort requires O
?np
?
time, whereas sorting floating point
numbers using merge sort requires O

time. Steps (2), (5), and (7) call the communication primitives transpose, bcast, and transpose, respectively. The analysis of these primitives in [6] shows that these three steps require

, respectively. Hence, with high probability, the overall complexity of our sorting algorithm is given (for floating point numbers) by

Clearly, our algorithm is asymptotically optimal with very small coefficients. But a theoretical comparison of our running time with previous sorting algorithms is difficult, since there is no consensus on how to model the cost of the irregular communication used by the most efficient algorithms.

Hence, it is very important to perform an empirical evaluation of an algorithm using a wide variety of benchmarks, as we will do next.

4 Performance Evaluation

Our sample sort algorithm was implemented using Split-C [10] and run on a variety of machines and processors, including the Cray Research T3D, the IBM SP-2-WN, and the Thinking Machines CM-5. For every platform, we tested our code on nine different benchmarks, each of which had both a 32-bit integer version (64-bit on the Cray T3D) and a 64-bit double precision floating point number (double) version.

4.1 Sorting Benchmarks

Our nine sorting benchmarks are defined as follows, in which n and p are assumed for simplicity but without loss of generality to be powers of two and MAXD, the maximum value allowed for doubles, is approximately 1:8 ? 10308.

1. Uniform [U], a uniformly distributed random input, obtained by

calling the C library random number generator random(). This function, which returns integers in the range to ?231 ? 1?, is
seeded by each processor Pi with the value (21+1001i). For the double data type, we \normalize" the integer benchmark values by first subtracting the value 230 and then scaling the result by ?2?30 ?MAXD ?.

2. Gaussian [G], a Gaussian distributed random input, approximated by adding four calls to random() and then dividing the result by four. For the double data type, we normalize the integer benchmark values in the manner described for [U].

3. Zero [Z], a zero entropy input, created by setting every value to a constant such as zero.

4. Bucket Sorted [B], an input that is sorted into p buckets, obtained by setting the first np2 elements at each processor to be random numbers between and (231
p ? 1), the second np2
elements at each processor to be random numbers between 231
p and (232
p ? 1), and so forth. For
the double data type, we normalize the integer benchmark values in the manner described for [U].

5. g-Group [g-G], an input created by first dividing the processors into groups of consecutive processors of size g, where g can be any integer which partitions p evenly. If we index these groups in consecutive order from 1 up to pg , then for group j we set the first npg elements to be random numbers between ???(j ? 1) g + p2 ? 1? mod

,
the second npg elements at each processor to be random numbers between

, and so forth.
For the double data type, we normalize the integer benchmark values in the manner described for [U].

6. Staggered [S], created as follows: if the processor index i is less than or equal to p2 , then we set all np elements at that processor to be random numbers between
?

. Otherwise, we set all np elements to be random numbers between

and
For the double data type, we normalize the integer benchmark values in the manner described for [U].

7. Worst-Load Regular [WR] - an input consisting of values between and (231 ?1) designed to
induce the worst possible load balance at the completion of our regular sorting. Specifically, at the completion of sorting, the

odd-indexed processors will hold (np +ns ?p) elements, whereas the evenindexed processors will hold (np ? ns +p) elements. The benchmark is defined as follows. At processor P1, for odd values of j between 1 and (p?2), the elements with indices

are set to random values between

. For the double data type, we normalize the integer benchmark values in the manner described for [U].

8. Deterministic Duplicates [DD], an input of duplicates in which we set all np elements at each of the first p2 processors to be log n, all np elements at each of the next p4 processors to be log ?n2 ?, and so forth. At processor Pp, we set the first n2p elements to be log ?np ?, the next n4p elements to be log ? n2p ?, and so forth.

9. Randomized Duplicates [RD], an input of duplicates in which each processor fills an array T with some constant number range (range is 32 for our work) of random values between and (range?1) whose sum is S. The first T [1] S ? np values of the input are then set to a random value between and (range ? 1), the next T [2] S ? np values of the input are then set to another random value between and (range ? 1), and so forth.

See [14] for a detailed justification of these benchmarks.

4.2 Experimental Results

For each experiment, the input is evenly distributed amongst the processors. The output consists of the elements in non-descending order arranged amongst the processors so that the elements at each processor are in sorted order and no element at processor Pi is greater than any element at processor Pj , for all i < j.

Two variations were allowed in our experiments. First, radix sort was used to sequentially sort integers, whereas merge sort was used to sort double precision floating point numbers (doubles). Second, different implementations of the communication primitives were allowed for each machine. Wherever possible, we tried to use the vendor supplied implementations. In fact, IBM does provide all of our communication primitives as part of its machine specific Collective Communication Library (CCL) [7] and MPI. As one might expect, they were faster than the high level Split-C implementation.

Optimal Number of Samples s for Sorting on T3D
Number of Processors

Table I: Optimal number of samples s for sorting the [WR] integer benchmark on the Cray T3D, for a variety of processors and input sizes.

Table II: Optimal number of samples s for sorting the [WR] integer benchmark on the IBM SP-2-WN, for a variety of processors and input sizes.

Tables I and II examine the preliminary question of the optimal number of samples s for sorting on

the Cray T3D and the IBM SP-2-WN. They show the value of s which achieved the best performance on the Worst-Load Regular [WR] benchmark, as a function of both the number of processors p and the number of keys per processor np . The results suggest that a good rule for choosing s is to set it to 2b 12 log(n=p)c ss
qnp , which is what we do for the remainder of this discussion. To compare this choice for s with the theoretical expectation, we recall that the complexity of Step (3) is O(sp log p), whereas the complexity of Step (9) is O
??np + ns ? p
?
log p
?
. Hence, the first term is an increasing function of
s, whereas the second term is a decreasing function of s. It is easy to verify that the expression for the sum of these two complexities is minimized for s = O
?qnp
?
, and, hence, the theoretical expectation
for the optimal value of s agrees with what we observe experimentally.

Table III: Total execution time (in seconds) required to sort a variety of integer benchmarks on a 64-node Cray T3D.

Table IV: Total execution time (in seconds) required to sort a variety of integer benchmarks on a 64-node IBM SP-2-WN.

Table V: Total execution time (in seconds) required to sort a variety of double benchmarks on a 64-node Cray T3D.

Tables III, IV, V, and VI display the performance of our sample sort as a function of input distribution for a variety of input sizes. In each case, the performance is essentially independent of the input distribution. These figures present results obtained on a 64 node Cray

T3D and a 64 node IBM

Table VI: Total execution time (in seconds) required to sort a variety of double benchmarks on a 64-node IBM SP-2-WN.

SP-2; results obtained from other platforms validate this claim as well. Because of this independence, the remainder of this section will only discuss the performance of our sample sort on the Worst-Load Regular benchmark [WR].

The results in Tables VII and VIII together with their graphs in Figure 1 examine the scalability of our sample sort as a function of machine size. Results are shown for the T3D, the SP-2-WN, and the CM-5. Bearing in mind that these graphs are log-log plots, they show that, for a given input size n, the execution time scales inversely with the number of processors p for (p <= 64). While this is certainly the expectation of our analytical model for doubles, it might at first appear to exceed our prediction of an O
?np log p
?
computational complexity for integers. However, the appearance of an inverse relationship is still quite reasonable when we note that, for values of p between 8 and 64, log p varies by only a factor of two. Moreover, this O
?np log p
?
complexity is entirely due to the merging
in Step (9), and in practice, Step (9) never accounts for more than 30% of the observed execution time. Note that the complexity of Step (9) could be reduced to O
?np
?
for integers using radix sort,
but the resulting execution time would, in most cases, be slower.

Regular Sorting of 8M Integers [WR]
Number of Processors
Machine 8 16 32 64 128

Table VII: Total execution time (in seconds) required to sort 8M integers on a variety of machines and processors using the [WR] benchmark. A hyphen indicates that particular platform was unavailable to us.

However, the results in Tables VII and VIII together with their graphs in Figure 1 also show that for p greater than 64, the inverse relationship between the execution time and the number of processors begins to deteriorate. Table IX explains these results with a step by step breakdown of the execution times reported for the sorting of integers on the T3D. Step (1) clearly displays the O ?np
?
complexity
expected for radix sort, and it dominates the total execution time for

small values of p. The transpose operation in Step (2) displays the
?
o + np oe
?
complexity we originally suggested. The dependence

Table VIII: Total execution time (in seconds) required to sort 8M doubles on a variety of machines and processors using the [WR] benchmark. A hyphen indicates that particular platform was unavailable to us.

Figure 1: Scalability of sorting integers and doubles with respect to machine size.

of o on p simply becomes more pronounced as p increases and np decreases. Step (3) exhibits the O(sp) complexity we anticipated, since for 2b 12 log(n=p)c, s is halved every other time p is doubled. Steps (6) and (9) display the expected O ?p2 log p? and O

?
complexity, respectively. Steps (7) and (8) exhibit the most complicated behavior. The reason for this is that in Step (7), each processor must exchange p subsequences with every other processor and must include with each subsequence a record consisting of four integer values which will allow the unshuffling in Step (8) to be performed efficiently. Hence, the O ? np2 + nsp + 4p
?

transpose block size in the case of 128 processors is nearly half that of the the case of 64 processors (1280 vs. 2816). This, together with the fact that o increases as a function of p, explains why the time required for Step (7) actually increases for 128 processors. Step (8) would also be expected to

Step by Step Breakdown of Sorting 8M Integers
Number of Processors (Number of Samples)

Table IX: Time required (in seconds) for each step of sorting 8M integers on the Cray T3D using the [WR] benchmark.

complexity. But the scheme chosen for unshuffling
also involves an O(p) amount of overhead for each group of p subsequences to assess their relationship so that they can be efficiently unshuffled. For sufficiently large values of p, this overhead begins to dominate the complexity. While the data of Table IX was collected for sorting integers on the T3D, the data from the SP-2-WN and the T3D support the same analysis for sorting both integers and doubles.
The graphs in Figure 2 examine the scalability of our regular sample sort as a function of keys per processor
?np
?

, for differing numbers of processors. They show that for a fixed number of up to 64 processors there is an almost linear dependence between the execution time and np . While this is certainly the expectation of our analytic model for integers, it might at first appear to exceed our prediction of a O
?np log n
?
computational complexity for floating point values. However, this appearance of a linear relationship is still quite reasonable when we consider that for the range of values shown log n differs by only a factor of 1:2. For p > 64, the relationship between the execution time and and np is no longer linear. But based on our discussion of the data in Table IX, for large p and relatively small n we would expect a sizeable contribution from those steps which exhibit O ?p2 log p?, O

complexity, which would explain this loss of linearity.
Finally, the graphs in Figure 3 examine the relative costs of the nine steps in our regular sample sort algorithm. Results are shown for both a 64 node T3D and a 64 node SP-2-WN, using both the integer and the double versions of the [WR] benchmark. Notice that for n = 64M integers, the sequential sorting, unshuffling, and merging performed in Steps (1), (8), and (9) consume approximately 85% of the execution time on the T3D and approximately 75% of the execution time on the SP-2. By contrast, the two transpose operations in Steps (2) and (7) together consume