Load Balanced Parallel Radix Sort

Andrew Sohn
Computer Information Science Dept.
New Jersey Institute of Technology
Newark, NJ 07102-1982
sohn@cis.njit.edu

Yuetsu Kodama
Parallel Computer Architecture Laboratory
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, Ibaraki 305, Japan
kodama@etl.go.jp

Abstract

Radix sort suffers from the unequal number of input keys due to the unknown characteristics of input keys. We present in this report a new radix sorting algorithm, called *balanced radix sort* which guarantees that each processor has exactly the same number of keys regardless of the data characteristics. The main idea of balanced radix sort is to store any processor which has over n/P keys to its neighbor processor, where n is the total number of keys and P is the number of processors. We have implemented balanced radix sort on two distributed-memory machines IBM SP2-WN and Cray T3E. Multiple versions of 32-bit and 64-bit integers and 64-bit doubles are implemented in Message Passing Interface for portability. The sequential and parallel versions consist of approximately 50 and 150 lines of C code respectively including parallel constructs. Experimental results indicate that balanced radix sort can sort **0.5G** integers in **20** seconds and 128M doubles in 15 seconds on a 64-processor SP2-WN while yielding over 40-fold speedup. When compared with other radix sorting algorithms, balanced radix sort outperformed, showing two to six times faster. When compared with sample sorting algorithms, which are known to outperform all similar methods, balanced radix sort is 30% to 100% faster based on the same machine and key initialization.

1 Introduction

Parallel sorting is one of the important components in parallel computing. As parallel computing becomes ubiquitous, the demand of efficient parallel sorting ever increases. There are a large number of parallel sorting algorithms developed, including Batcher's bitonic sorting, flash sorting, radix sorting, parallel merge sorting, sample sorting, etc. Among the issues in parallel sorting are communication and data characteristics. Parallel sorting algorithms need to address the two issues if they are to be successful for large problems on large-scale parallel machines. Blelloch, et al. and Dusseau, et al. present comparative studies of different parallel sorting algorithms on CM-2 [3] and CM-5 under LogP [5,6], respectively.

Parallel sorting dates back to Batcher's bitonic sorting [2]. Bitonic sorting consists of two steps: local sort and merge. Given a locally sorted list, processors keep merging across processors in pairs. The major advantage of bitonic sorting is its simplicity in communication. Since all processors work in pairs, the communication pattern becomes very simple throughout the entire computation. However, bitonic sorting can suffer from long communication time depending upon the data characteristics. It is often the case that the entire keys assigned to each processor need to be exchanged due to data characteristics. Should this occur, the long communication time will result when the number of keys is realistically large. Bitonic sorting can also become complicated due to the way the keys are kept on each processor. Half of the processors can keep the keys in an ascending order while the remaining half will keep in a descending order. Or, keys can always be kept in an ascending order, in which case the way processors communicate will change. Regardless of its complexity in implementation, bitonic sorting can be a choice due to its simplicity in communication.

In Proceedings of the 12th ACM International Conference on Supercomputing, Melbourne, Australia, July 14-17, 1998.

Sample sort [7] has recently attracted much attention in parallel sorting [12,10,8,9]. Sample sorting works by picking some splitters randomly. These random splitters are used to separate the keys into buckets. Those keys within each bucket are then locally sorted. Some load balancing steps may be necessary since buckets may not necessarily have the same or a similar number of keys. Random sample sorting thus depends heavily on how these splitters are picked. To avoid these random splitters, samples are picked regularly from the sorted list, called regular sample sorting. Gerbessiotis and Siniolakis studied sample sorting using the BSP model [8,14]. Helman, Bader, and JáJá, indicated that regular sample sort seems to outperform all similar methods [9]. A drawback of sample sorting is that it can be complicated since picking splitters and division of keys involve somewhat complex procedures [3].

Radix sort is often used on parallel machines due to its simplicity in implementation. The idea behind radix sort is bin sorting to treat processors as bins. Scanning a radix of some bits, keys are stored at the corresponding bins (or processors). Radix sort typically involves four steps in each round of radix: bin counting, local moving, global prefix sum of bin counts, and communication. The main problem with radix sort however is its irregularity in computation and communication. Since data characteristics are usually unknown a priori, it is not clear how much computation and communication each processor will take. Second, it is unlikely that processors receive a similar number of keys due again to the characteristics of input keys.

It is the purpose of this report to introduce a new parallel radix sorting algorithm, called *load balanced radix sorting*. The balanced radix sorting guarantees that each processor has exactly the same number of keys, thus eliminating the load balancing problem. The idea is to first obtain the bin counts of all the processors, and then compute which processors get how many keys from what bins and what processors. Overloaded processors will spill keys to their immediate neighbors. This computation is done with *one* all-to-all bin count transpose operation. For 32-bit integers with the radix of 8 bits, balanced radix sort will need 4 all-to-all transpose operations of bin count. Keys will be moved after all the bins and their keys are located in the global processor space.

The paper is organized as follows. In section 2, we define the load balancing problem of parallel radix sort. Section 3 presents the key idea of balanced parallel radix sort. Section 4 gives implementation details and experimental results based on IBM SP2-WN and Cray T3E multiprocessors. Absolute performance of balanced radix sort is presented along with five initialization methods. In section 5, we compare the performance of balanced radix sort with other sorting methods. The last section concludes this report.

2 Load Imbalance in Parallel Radix Sort

Radix sorting is a very simple and yet powerful in terms of both logic and implementation. Each processor locally scans keys using some predefined number of bits to find local bin count. The keys are scanned again and moved to appropriate bins within each processor according to the bin count. Upon completion of this local computation, keys of the same bins from all the processors are collected into a designated processor. This completes an iteration of simple radix sort. The major problem with parallel radix sort, however, is the load balancing problem. Due to the unknown characteristics of input keys, some processors will have a lot of keys while some have a few. The imbalance in the number of keys will result in unbalanced computations and in turn irregular communication. The performance of parallel radix sort, therefore, is limited by the degree of the load imbalance due to the characteristics of input keys.

Before we proceed to the load balance problem in radix sorting, we list below the symbols used in this report:

- P is the number of processors.
- n is the problem size, i.e., the total number of keys.
- g is the group of bits for each scanning (or round).
- r is the number of rounds each key goes through. Rounds and passes will be used interchangeably.
- b is the number of bits for integer/doubles. We consider 32-bit and 64-bit integers. Doubles are always 64 bits.
- B is the number of buckets (bins) for the given group of bits, i.e., $B = 2^g$. Buckets and bins will be used interchangeably.

Serial radix sorting works as follows: Given the numbers of b bits, a group of g consecutive bits is scanned from the least significant bit. Keys are stored in 2^g buckets according to the g bits. For b bit integers, radix sort requires $r = b/g$ rounds (passes) to sort the entire keys. To be more precise, each round consists of two steps: *count* and *move*. The count step first counts how many keys each bucket has. This step identifies the global map of the keys for the given bits by computing exclusive prefix sum of each bucket. Each bucket will then be assigned a starting address according to the exclusive prefix sum. The move step then examines the g bits of each key to determine a bucket number. Given the bucket number, each key is assigned a relative location within the bucket. The key is then actually moved to that location to complete the move step.

Parallel radix sort is not much different from sequential radix sort. The only difference is that the keys are stored across processors. Each processor can hold one to many buckets. Given P processors and g bits, each processor will hold B/P buckets. To simplify the discussion we assume B=P. A simple parallel radix sorting consists of four steps:

1. **Count**: count the number of keys in each bucket by scanning keys using g bits (local operation).
2. **Move**: move within each processor the keys to an appropriate bucket by re-scanning all the keys (local operation).
3. **Transpose**: *1-to-all* transpose the bucket information across processors to find the prefix sum (global operation).
4. **Communication**: send/receive keys to/from the destination/source processors.

Let us consider sorting 40 keys on four processors, P0..P3. The 40 keys are initially distributed equally to four processors, each of which holds 10 keys. Assume two bits are scanned in each round, requiring four buckets B0..B3. Assume further that each processor scanned the keys and moved them to appropriate buckets. Figure 1 shows a snapshot of the bins after the first two steps of count and move. The snap shot indicates that the processors each scanned 10 keys and computed bin-counts. Processor 0 has four bins with the bin counts of 1, 3, 4, and 2. P1 has four bins with the bin counts of 3, 6, 1, and 0. Other processors have done similar bin counting.

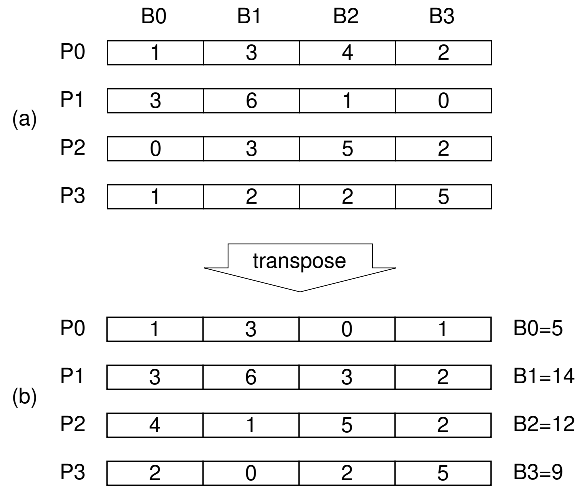

Figure 1: A round of typical radix sorting, consisting of bin count and transpose the bin counts across processors.

Now that each processor has its own bin counts, the third step of a 1-to-all transpose operation is applied to all four processors to obtain the global map of bin counts. After transposing the bin counts across the four processors, each processor obtains *four* counts of the *same* bin. Processor 0 collects four B0's from four processors, including itself, P1 collects four B1's, etc. Processor 0 finds that it has 5 keys in B0, P1 has 14 keys in B1, etc.

In the last step, processors send/receive keys to appropriate processors according to the global key map. The four processors will first send keys according to the map shown in Figure 1(a). Processor 0 sends 3 keys of Bin1 to P1, 4 keys of Bin2 to P2, and 2 keys of Bin3 to P3. Other processors do similar send operations. When sending (or posting of send operations) is complete, processors will receive keys according to the map shown in Figure 1(b). Processor 0 will receive 3 keys from P1, nothing from P2, and 1 key from P3. When the keys are received, they will be stored according to the rank of the source processor, assuming that processor ranks are preserved throughout the computation. Sending and receiving can be done in any order as long as they are done correctly and efficiently.

When the first round is complete, the four processors now each have a different number of keys. Processor 0 has only 5 keys, which are half the original size. Processor 1 has 14 keys, 40% increase in size. Processor 2 has 12 keys, 2 more than before. P3 has 9 keys, one less than before. These varying numbers of keys will cause heavy load imbalance both in computation and communication in the second round. Bin counting and moving 14 keys will certainly take more than twice the time taken for 5 keys. Sending and receiving yet undetermined numbers of keys will cause imbalance in communication. The total execution time will therefore be determined by the processor that executes the critical path, i.e., the most keys. Unfortunately, the characteristics of keys are often unknown for real-world problems. Radix sorting based on the above method will give poor performance as there is no mechanism to guarantee an even distribution of keys across processors.

3 Load Balanced Parallel Radix Sort

Balanced radix sort is designed to eliminate the load imbalance caused by the characteristics of keys. The nature of keys is no longer the bottleneck of parallel radix sort. To illustrate the new balanced radix sort, we revisit the four steps of unbalanced radix sort and list below with a slight modification:

- Count the number of keys in each bucket by scanning g bits.
- Move locally the keys to an appropriate bucket by re-scanning.

- *All-to-all transpose* the bin count information across processors. Each processor has the bin count of all processors.
- Send/receive the keys with its location information to the destination processors.

The first two steps of count and move are the same. The major difference between the unbalanced and the balanced radix sort is in the third step. The fourth step will be modified slightly to reflect the change in the third step. The transpose operation now is all-to-all. *Each processor will have a complete map of bin counts across the processors*. In the unbalanced radix sort, it was sufficient for each processor to have the same buckets collected from all other processors. Balanced radix sort, however, will have a complete bin count of all processors. Based on this complete map of bin count, each bin and their keys can be precisely located across processors. The sending/receiving operations will be modified according to this new bin count map. Figure 2 illustrates how this can be achieved. We continue to use the example shown in Figure 1, where four processors sort 40 keys.

Figure 2(a) shows the bin count after an all-to-all transpose operation. Now, this map is kept on all processors. (Figure 2(a) is the same as Figure 1(b).) Figure 2(b) is a load balanced map based on the bin count shown in Figure 2(a). The thick lines indicate how the bins will be split to generate an equal number of keys to each processor. Each processor eventually gets the same number of keys, as shown in Figure 2(c).

Given the global map of bin counts shown in Figure 2(a), each processor determines how it will collect the keys needed for itself. It does so by simply scanning the map starting B0 and identifies how many keys are from what bins. In the above example, processor 0 determines that it will need six bins to make 10 keys. It not only needs the four B0's collected from the four processors but also extends its key gathering to two B1's. The four B0's give only 6 keys which are not sufficient to make 10 keys. P0 will therefore gather 3 keys from B1 of its own and 2 keys from B1 of processor 1. The key gathering decision spans six buckets over four processors, as shown in Figure 2(c).

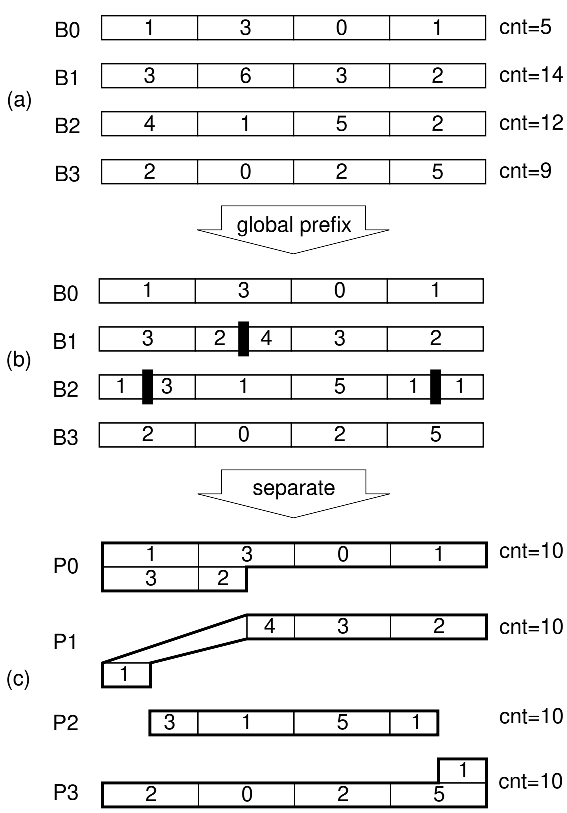

Figure 2: Converting the global count map to a load balanced map. The map shown in (a), which is resulted from an all-to-all transpose operation, is kept on all processors. The thick lines in (b) show how bin counts will be split to achieve a balanced number of keys.

Processor 1 determines to gather 10 keys from three B1's and one B2, as shown in Figure 2(c). It gathers four keys from B1 of its own, 3 keys from B1 of P2, 2 keys from B1 of P3, and 1 key from B2 of P0. The key gathering decision for P1 spans four buckets over four processors. Processors 2 and 3 similarly determine to gather 10 keys each. P2 finds 10 keys from B2 of P0, P1, and P2. Processor 3 determines to gather 10 keys from B2 of P3 and four B3's from four processors. This step completes the decision on how keys will be collected from various buckets and processors.

The last step will actually send or receive keys according to the decision made in the extended prefix computations. It may appear that the address computation is complex. And the sending/receiving can further complicate since much information is needed to keep track of which part of buckets to send to where with how many keys. However, all this information is already stored in the global count map. Computing and keeping this information is straightforward as the experimental results will demonstrate.

The global all-to-all transpose operation gives each processor the complete information of bin counts. As we will demonstrate shortly, this all-to-all transpose is trivial for reasonable data size since we are collecting bin counters, not the actual keys. Given the complete map of bin counts, each processor identifies the necessary bins to collect n/P keys. This computation is also trivial once the global count map is obtained.

4 Experimental Results and Discussion

4.1 Some implementation details

Balanced radix sort has been implemented on two distributed-memory multiprocessors IBM SP2 Wide Node (WN) installed at NASA Ames and Langley and Cray T3E installed at NERSC, Lawrence Berkeley Laboratory. For each machine there are two versions; SP2 versions have 32-bit integer and 64-bit double. T3E versions include 64-bit integer and 64-bit double. All these versions are somewhat different because of the way numbers are represented. For integer sorting, we used 8 bits in each round, requiring four rounds for 32-bit integers on SP2 and eight rounds for 64-bit integers on T3E. Doubles are 64 bit numbers, consisting of 1 bit for sign, 11 bits for exponent, and 52 bits for mantissa. We used 8 bits in each round for the low 48 bits and 4 bits for the remaining 4 bits of mantissa. The total of 7 rounds is required for mantissa. Exponents are split into two rounds: 8 bit radix for low exponent bits and 4 bit radix for high exponent bits.

Sorting often requires two to three buffers of the same data size for various purposes. The first one is used for storing the original keys. The second is used for storing immediate keys when sorting is performed. The third one would be necessary for communication, i.e., used as a communication buffer. Our implementation, however, uses only two buffers. The first one is used to store keys and the second for buffering. Since our implementation required only two buffers, each processor can sort up to 8M integers and 4M doubles. The maximum data size of over 64 processors is 512 M integers and 256M doubles. This data size is significantly larger than the Split-C implementation reported in [9] where the maximum of up to 1M integers per processor is used. Another reason we were able to sort up to 8M integers per processor is that our implementation is portable based on Message Passing Interface (MPI) which requires no special programming is runtime environment.

All of the above have been implemented in MPI for portability. The programs are very compact and simple. Sequential radix sorting is about 50 lines of C code. Balanced parallel radix sorting is about 150 lines of C code which include MPI constructs. These code sizes do not include array initialization and self-checking routines. The programs do not require any special features or special runtime environment. If the machine supports MPI, the sorting routines will work with no modifications.

The keys are initialized with the five different methods used in[9], including random, gauss, zero, bucket, and stagger. Random simply calls the C random() routine which initializes the full 32 bits for integers. Doubles (64-bit) are split into two 32-bits, each of which is initialized separately. Therefore, all the 64 bits of doubles are fully initialized. Gauss initializes by adding the results of four random() calls and dividing it by four. Zero is essentially the same as random except some keys set to zero. We set 10% of keys to zero in each processor. Bucket and stagger initialization methods are based on the methods described in [9]. These methods are designed to create some artificial characteristics of keys. There are other initialization methods but we decided not to explore these variations. As we shall see below, the balanced radix sort is not highly sensitive to the characteristics of keys. We therefore find that the five different initialization methods suffice to demonstrate the absolute and relative performance of balanced parallel radix sort.

4.2 Absolute performance

Tables 1 and 2 list some execution results on 64 processors. The results show the relation between data size and various initialization methods for integers and doubles. They are intended to give a feel for the performance of the balanced radix sort algorithm. The integer results are listed under five different initialization methods. Doubles used the gauss initialization method. Table 1 lists SP2-WN results while Table 2 lists T3E results.

# of integers	SP-2, 64 processors, **32**-bit integers					**64**-bit doubles
	random	gauss	zero	bucket	stagger	gauss
1M	0.137	0.135	0.136	0.116	0.136	0.313
2M	0.181	0.182	0.179	0.170	0.172	0.456
4M	0.264	0.254	0.252	0.212	0.210	0.684
8M	0.399	0.395	0.414	0.330	0.336	1.164
16M	0.684	0.678	0.750	0.560	0.606	2.120
32M	1.142	1.167	1.308	1.148	1.298	4.064
64M	2.178	2.225	2.477	2.286	2.665	7.869
128M	4.272	4.465	4.905	4.598	5.524	15.611
256M	9.250	9.485	9.601	10.000	12.072	-
512M	18.673	19.427	18.900	19.656	24.581	-

Table 1: Execution times (sec) on a 64-processor SP2-WN.

# of integers	Cray T3E, 64 processors, 64-bit integers					64-bit doubles
	random	gauss	zero	bucket	stagger	gauss
1M	0.221	0.228	0.191	0.180	0.223	0.220
2M	0.301	0.317	0.261	0.236	0.266	0.307
4M	0.413	0.447	0.364	0.326	0.337	0.450
8M	0.497	0.575	0.464	0.425	0.455	0.576
16M	0.632	0.758	0.612	0.581	0.647	0.752
32M	1.087	1.222	1.057	1.082	1.212	1.314
64M	2.055	2.217	1.944	2.042	2.255	2.518
128M	3.975	4.084	3.736	3.919	4.458	4.904
256M	7.598	7.693	7.312	7.616	9.052	-
512M	18.474	19.787	17.715	18.270	20.445	-

Table 2: Execution times (sec) on a 64-processor T3E.

Table 1 shows that SP2 can sort 512M integers in less than 25 seconds. Most of the initialization methods give similar results with little variations, except the stagger method. Our results agree with the results reported by Helman, Bader, and JáJá in [9], where stagger gave the worst results. When comparing the results of integers and doubles, we find that there is a substantial difference. It is obvious because SP2 is a 32-bit machine. Doubles will take at least twice the time as integers. In fact, it takes almost three times the integer results because of cache effects. As data size becomes larger, this effect becomes more apparent.

T3E results shown in Table 2 are slightly better than SP2 for integers in general. It is especially true when the data size is large. The reasons are the faster clock and the faster network architecture of T3E. SP2 incurs much overhead due to message passing. The SP2 latency is typically over 40 μsec while the T3E latency is a few to 10 μsec. This rather large difference in latency does make a difference on large data size as seen from the two tables. When comparing the integer and double results for T3E, we find they are essentially the same. Since T3E is a 64-bit machine, the results must be very similar regardless of integers or doubles.

The difference between Gauss and Stagger for T3E is much smaller than that for SP2. For SP2 in Table 1, the ratio of Stagger to Gauss is 24.581/19.427 = 1.27, or 27%. However, for T3E, it is 20.445/19.187 = 1.07, or 7%. This difference in ratio is due to the fact that fast communication can help tolerate the irregular data characteristics of Stagger.

Figure 3 shows the relation between the number of processors and initialization methods. The figure indicates that the effect of initialization methods diminishes as the number of processors is increased. The plots also suggest that balanced radix sort is scalable, which is discussed below.

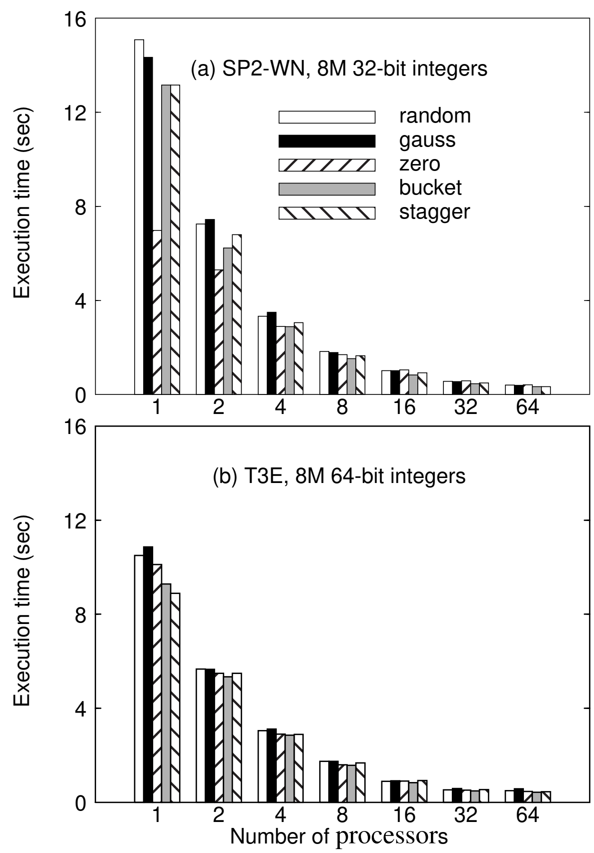

Figure 3: Execution times (sec) of 8M integers.

4.3 Scalability of Balanced Radix Sort

The plots in Figure 4 demonstrate the scalability of balanced radix sort. The results are based on Gauss initialization. We were able to sort up to 8M integers on a *single* processor. Therefore, the results for 1M to 8M are compared against a single processor performance.

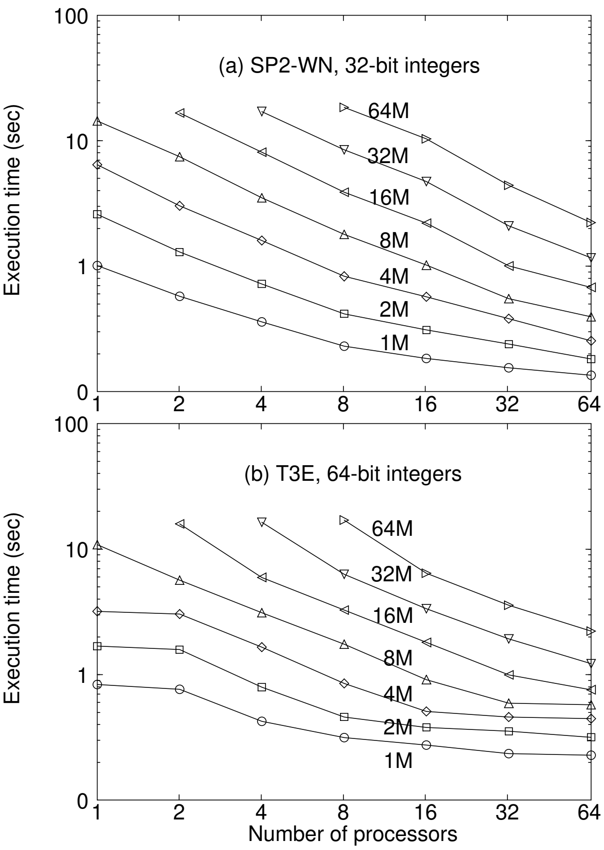

Figure 4: Execution times (sec) using Gauss initialization.

For the data size of up to 8M, SP2 shows over 40-fold speedup. However, T3E gives only 20-fold speedup on 64 processors. The two machines show a significant difference in scalability. The reason T3E shows half the scalability of SP2 is because of its MPI implementation. T3E is designed to use SHMEM programming environment to exploit the underlying architecture such as External registers and stream features for fast remote memory operations.

Specifically, there are two reasons why T3E shows low scalability when implemented in MPI. One is the large latency due to MPI implementation. To help understand this low scalability, we list in Table 3 some machine characteristics.

Programming paradigm	Cray T3E		IBM SP2	
	Latency	Bandwidth	Latency	Bandwidth
SHMEM put	1.3	336	-	-
SHMEM get	1.5	336	-	-
MPI	12.8	108	40	45

Table 3: A brief comparison of the environments [15,11]. Latency is in µsec and bandwidth in MBytes/sec.

As we note from the table, the MPI latency is about 10 times more than the SHMEM one [15]. The benchmark study on different programming paradigms indicated that the native SHMEM remote memory operation Put incurs 1.3 µsec. On the other hand, MPI Send incurs 12.8 µsec. The two latencies are different by an order of magnitude! The reason such a large difference is that SHMEM Put is one-sided communication while MPI Send is two-sided communication. Depending on the low-level optimization, the MPI construct requires the attention of two processors for eventual handshaking. However, one-sided communication needs only one processor's attention. The rest is often left to the programmer. While MPI can use non-blocking constructs which can help free the processors from locking, this can also be left to programmers.

The second reason T3E shows low scalability is that the MPI implementation does not adequately exploit the low-level hardware features. Together with the External registers and the stream feature, SHMEM can exploit fine-grain communication. However, MPI can be problematic when the communication is done in small data size. For every send/receive operation, it requires an overhead of copying the data to buffer, sending out, and confirming. Communication studies show that fine-grain communication on SP-2 with MPI is inefficient [13]. The study also indicated that there is a threshold of how small the message size should be and how many messages there should be for efficient communication. SP2 with MPI is efficient when the message size is between 4KB to 16KB.

When the MPI version is translated to SHMEM version to take advantage of the fine-grain one-sided communication features, we expect that T3E will outperform the MPI implementation. This issue is beyond the scope of this report and will be addressed in the future. As we have emphasized earlier, our efforts are expended on portability, being able to run on a variety of machines, not a specific machine. Therefore, there is a trade-off between the native programming implementation and the portable implementation.

4.4 Distribution of execution times

Identifying where the total execution times are spent, we will be able to better understand how balanced radix fared. Recall that balanced radix sort consists of four steps: local count, local move, all-to-all transposition of bin counter, and communication of keys. Figure 5 illustrates how the executions times are spent on these four steps. From the bottom to the top are count, move, transpose, and communication. The top two entries are communication times and the bottom two entries are local computation times. The plots are for 64 processors. The x-axis shows the problem size and the y-axis the percentage of an individual execution time.

There are several observations we make from the bar charts: First, computation is small for both machines while communication dominates. However, as the problem size is increased, the computation time does proportionally increase. Second, the all-to-all transpose time is substantial (10 to 20%) for small problem size of 1M to 4M. However, for reasonably sized problems, the time is negligible. In fact, when the data size is over 32M, the time becomes a small percent of the total sorting time. Third, despite the small transpose time, the overall communication times (the two top entries of each bar) remain relatively constant across different machines and data type. This clearly indicates the nature of sorting which states that sorting is communication intensive.

Since computation is very simple, it will not be straightforward to further reduce the computation time. However, since communication dominates the overall sorting time, the next step for developing fast sorting algorithms needs to expend efforts in reducing the communication time. In this study, we have not attempted to optimize the communication part of sorting as our code size indicates (parallel sorting has approximately 150 lines of C code). We used the packaged parallel constructs included in MPI for all-to-all transpose. The sending and receiving operations are simply done by MPI constructs. We have made no efforts on message vectorization, nor personalized communication since this will defeat the purpose of simple parallel radix sorting. Our next step therefore is to look into some possible improvements in communication.

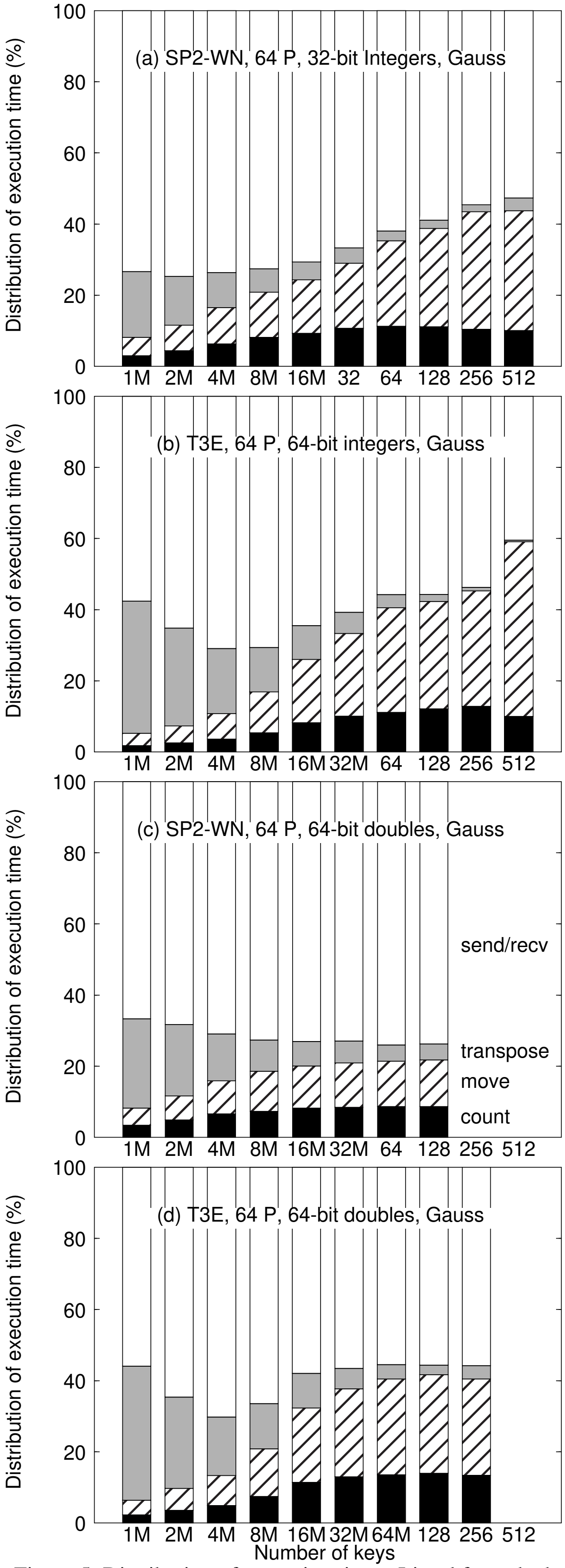

Figure 5: Distribution of execution times. Listed from the bottom are local bin count, local key move, global all-to-all counter transpose, and global sending/receiving of keys.

5 Comparison with Other Sorting Methods

The performance of balanced radix sort is compared against three sorting methods: conventional unbalanced radix sort, regular sample sort, and random sample sort. The comparisons are based on the results published in [1,9]. We believe the comparisons are fair because the initialization methods and machines are the same. Table 4 compares the results of the three different radix sortings on a 16-processor IBM SP2-WN. All the results use the same machine and the same initialization methods [9].

init method	size	Execution time (seconds)			Improvement	
		Balanced	HBJ [9]	AIS [1]	HBJ/Bal	AIS/Bal
Random	64K	0.076	0.107	0.474	1.4	6.2
	1M	0.185	0.592	0.938	3.2	5.1
	8M	1.016	4.030	4.130	4.0	4.1
	64M	10.198	n/a	n/a	-	-
Gauss	64K	0.073	0.109	0.475	1.5	6.5
	1M	0.184	0.613	0.907	3.3	4.9
	8M	1.019	4.120	4.220	4.0	4.1
	64M	10.358	n/a	n/a	-	-

Table 4: Comparison of three 32-bit integer radix sorting algorithms on a 16-processor SP2-WN. n/a=not available.

The results indicate that balanced radix sort outperformed the other two radix sorting by a factor of *three* to *five* for the data size of over 1M. When the data size is very small, as 64K on 16 processors, balanced radix sort still preforms better by 40% to 600%. For very large data size, the results of the other two sorting are not available for comparison. The main reason that our radix sort outperformed the other two radix sorts is because our method balances computation.

It should be noted that the other two radix sorts are not necessarily the best performing algorithms. It is thus not certain that the radix sorting methods are compared fairly. To avoid this possible unfairness, we compare balanced radix sorting with *random sample sorting* and *regular sample sorting* which were implemented in Split-C [4] and reported that they seem to outperform all similar algorithms [9].

Table 5 compares the performance of balanced radix sort with regular and random sample sorting on SP2-WN. The performance comparison on T3E was not possible because we were unable to find sample sort data on T3E. Note that balanced radix sorting used gauss initialization while sample sorting used WR initialization. Note further that the results of random sample sort using Gauss are essentially the same as those using WR: (gauss, WR) = (4.21, 4.22), (1.06, 1.07), (0.272, 0.269), and (0.701, 0.710) [Table 1 of ref. 9]. Therefore, we believe it is fair to compare Gauss results of balanced radix sort with WR results of sample sort.

	P=1	P=2	P=4	P=8	P=16	P=32	P=64	init
Balanced	13.153	6.231	2.881	1.52	0.83	0.455	0.330	Gauss*
Random	-	-	-	2.41	1.24	0.696	0.421	WR*
Regular	-	-	-	3.12	1.57	0.864	-	WR*
Ran/Bal	-	-	-	**1.6**	**1.5**	**1.5**	**1.3**	
Reg/Bal	-	-	-	**2.1**	**1.9**	**1.9**	**-**	

Table 5: Comparison on SP2-WN with 8M integers. *Gauss performs essentially the same as WR [9]. Ran = random sample sorting, Reg = regular sample sorting, Bal = balanced radix sorting.

The results indicate that balanced radix sort is overall substantially faster than sample sorting. When compared with random sample sorting, balanced radix sort is faster by 30% to 60%. When compared with regular sample sorting, balanced radix sorting is consistently faster by 100%. The main reason balanced radix sort is much faster than sample sorting is because balanced radix sort is simple and straightforward in terms of (a) the idea, (b) the implementation, and (c) its environment requirement. The idea of balanced radix sort is simple, requiring only four major steps: local count, local move, all-to-all counter transpose, and communication. Since the logic is simple, its implementation can also be made simple. The entire parallel algorithm has been implemented in approximately 150 lines of C code, including parallel constructs. This size does not include various key initialization and self checking. The third reason is balanced radix sort does not use any special programming environment, nor runtime systems like the Split-C implementation [9]. The only environment needed is an MPI library which is more or less becoming a standard environment for distributed-memory machines.

Sample sort, on the other hand, requires complex procedures. Two comparative studies on parallel sorting indicated that sample sort is the *most* complicated among several sorting algorithms including radix sort, column sort, sample sort, and bitonic sort [3,6]. Sample sort reported in [9] consists of nine steps, including local sort, sending/receiving keys, selecting splitters, sending/receiving splitters, rearranging splitters, etc. We list below a typical sample sort consisting of eight steps:

(1) Each processor locally sorts n/P keys, separating the sorted keys into P bins.
(2) Each processor sends bin j of size $0..n/P$ keys to processor j. The size of each bin can range 0 to n/P keys, depending on the key characteristics.
(3) Each processor receives $P-1$ sequences of keys and selects s samples. These s samples are sent to processor 0.
(4) Processor 0 collects from each processor a sequence of s samples. These P sequences of samples are merged to form a sequence of $s*P$ samples. P values are selected from these $s*P$ samples, called *splitters*.
(5) Processor 0 broadcasts the P splitters.
(6) Each processor receives the P splitters. Each sorted subsequence received in Step (2) is then rearranged based on the P splitters, resulting in yet another set of P subsequences. Each processor, therefore, generates the total of P^2 subsequences, where a set of P subsequences corresponds to each splitter.
(7) Each processor sends to processor j a set of P subsequences of *keys* that correspond to splitter j.
(8) Each processor receives P^2 subsequences of *keys* and merges them to result in a sorted list.

As it is clear from the above description, sample sort requires various local operations and global communications. Its communication steps include two all-to-all communication of *keys*, in addition to the all-to-all communication of samples and broadcasting of splitters.

To identify why balanced radix outperformed sample sort, we list in Table 6 the procedures involved in the two sorting algorithms. The table shows balanced radix sort for the radix of 8 bits for 32-bit integers. Balanced radix sort is much simpler compared to sample sort both in terms of computation and communication. Unlike sample sort, radix sort requires no step to be performed by a *single* processor. Therefore, there is no potential bottleneck which can hold all other processors. An all-to-all transpose operation of bin count information involves a very small number of integers sent to and received from all other processors. No complex procedures to select samples and splitters are needed. The only computation required, except for local count and move, is to find a global map for bin count. However, this procedure has approximately 20 lines of C code since all-to-all bin count communication performs the necessary computation for finding the map. This simplicity in computation and communication has directly contributed to the performance shown in Tables 4 and 5.

Balanced Radix Sort	Sample Sort
for (i=0; i<4; i++) {	local sort
local count and move	send/recv keys
all-to-all transpose (256 integers/prcr)	local sample selection
send/recv keys	1-to-all send/recv samples
}	(n/P^2 samples/processor)
	splitter selection on a single processor
	broadcasting splitters
	send/recv keys
	local merge

Table 6: Major steps in balanced radix sorting and sample sorting. Radix sort iterates 4 times for the radix of 8 bits for 32-bit integers.

6 Conclusions

Parallel radix sort is simple and straightforward. Its implementation can be as simple as a few tens of lines of C code if implemented properly. However, despite its simplicity in logic and implementation, parallel radix sort has suffered from the load balancing problem. Due to the characteristics of keys, some processors often receive a lot of keys while others do not. This load imbalance can be a critical issue when a large number of keys are to be sorted on a large number of processors. In this report, we have revisited the traditional radix sort and have presented a new parallel radix sort, called balanced radix sort that eliminates the load balancing problem. The main idea behind this new algorithm is to first obtain the bin count of all the processors, and then compute which processors get how many keys from what bin and what processors. Those overloaded processors will spill keys to their neighbor processors. Balanced radix sort guarantees exactly the same number of keys to all processors.

To verify the idea, we have implemented balanced radix sort on two distributed-memory multiprocessors: IBM SP2-Wide Node and Cray T3E. There are four different versions for each machine: sequential integer, parallel integer, sequential double, and parallel double. SP2 versions include 32-bit integer and 64-bit doubles. T3E versions include 64-bit integers and 64-bit doubles. The sequential versions are approximately 50 lines of C code while the parallel ones are approximately 150 lines of C code, without including array initialization and self-checking routines. All the parallel versions are implemented in Message Passing Interface. Since MPI does not require special programming environment, their code sizes are small. Due to its simplicity in implementation, we have been able to sort up to 8M integers on a single processor. In total, we have been able to sort 512M integers and 128M doubles on a 64-processor SP-2.

Experimental results have indicated that balanced radix sort can sort 512M integers in 20 seconds and 128M doubles in 15 seconds on a 64-processor SP2-WN. We have also identified that various array initialization methods do not give wide variations. For small problems size and number of processors, there were some variations. However, for large data size and number of processors, the variations due to different key initialization were essentially very small. The scalability of balanced radix sort has reached 40-fold speedup on a 64-processor SP2-WN and 20-fold speedup on a 64-processor T3E. The large difference between SP2 and T3E has been due to inefficient programming environment. T3E is designed specifically for the SHMEM programming environment that uses one-sided communication constructs such as put and get. However, the MPI implementation hobbled the capability of T3E since MPI incurs more than 10 times the latency and overhead of SHMEM. When translated to SHMEM on T3E, we believe the performance will reach SP2 scalability.

Balanced radix sort has been compared with other sorting results to identify the relative performance. Other methods include radix sort, random sample sort, and regular sample sort. When compared with other radix sorting algorithms on the same platform with the same initialization, balanced radix sort has simply outperformed, showing *two* to *five* times faster. Sample sorting has been known to outperform all similar parallel sorting methods. When compared with sample sorting algorithms, balanced radix sorting is 30% to 100% faster. The load balancing problem present in radix sorting seems to have been solved by the balanced radix sorting method presented in this paper. Radix sort can now be the choice for parallel sorting since it is simple, easy to code and maintain, requires a small amount of memory, and yet performs faster than any other results reported to date.

Our next step is to reduce the communication times that occupied over 60% of sorting time for large data size. There are several approaches we plan to undertake. The first approach is to incorporate message vectorization to reduce the excessive number of messages and at the same time increase the message size. If those messages that are from different buckets within each processor but destined to the same processor can be grouped together, it will significantly reduce the overhead associated with sending and receiving. The second approach is to convert the two-sided MPI communication constructs to one-sided MPI-2 constructs. This conversion will help save processor synchronization since processors in principle need not acknowledge each other for sending and receiving. However, this improvement depends greatly on the low-level implementations of MPI-2 one-sided communication constructs.

Acknowledgments

Andrew Sohn is supported in part by NSF INT-9722545, NSF INT-9722187, and the NASA JOVE Program NAG8 1114-2. Andrew Sohn would like to thank Horst Simon of NERSC, Lawrence Berkeley National Laboratory for much help and encouragement in computational science research. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Energy Research of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. Andrew Sohn sincerely thanks Joseph Oliger of Research Institute for Advanced Computer Science (RIACS) at NASA Ames for travel support and Doug Sakal of MRJ Technology Solutions at NASA Ames for summer support. The IBM SP-2s installed at the NASA Ames and Langley Research Centers were used to perform part of the experiments. Part of this work was performed while the author was visiting NASA Ames in the summer of 1997. Special thanks go to David Nassimi of NJIT for frequent discussions on parallel sorting. The authors thank the EM-X multithreaded distributed-memory multiprocessor group members, Mitsuhisa Sato, Hirofumi Sakane, Hayato Yamana, and Yoshinori Yamaguchi, of the Parallel Computer Architecture Laboratory of the Electrotechincal Laboratory of Japan for discussions on parallel sorting.

References

[1] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, LogGP: Incorporating long messages into the LogP Model - One step closer towards a realistic model for parallel computation. In *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, July 1995, pp.95-105.

[2] K. Batcher, Sorting networks and their applications, in *Proceedings of the AFIPS Spring Joint Computer Conference 32*, Reston, VA, 1968, pp.307-314.

[3] 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 *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, South Carolina, July 1991, pp.3-16.

[4] D. E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, Parallel Programming in Split-C, *In Proceedings of Supercomputing '93*, pages 262-273, Portland, OR, November 1993.

[5] D. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken, LogP: Towards a Realistic Model of Parallel Computation, in *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[6] A. C. Dusseau, D.E. Culler, K. Schauser, R. P. Martin, Fast parallel sorting under LogP: Experience with the CM-5, *IEEE Transactions on Parallel and Distributed Systems 7*, August 1996, pp.791-805.

[7] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM 17*, 1970, pp.496-507.

[8] A.V. Gerbessiotis and C.J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, June 1996, pp. 223-232.

[9] D.R. Helman, D.A. Bader, and J. JáJá. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, June 1996, pp.211-220.

[10] X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling, *Parallel Computing 19*, 1993, pp.1079-1103.

[11] NASA MetaCenter Home Page, Parallel systems documentation, http://parallel.nas.nasa.gov/Parallel/SP2/, NASA Ames Research Center.

[12] H. Shi and J. Schaeffer, Parallel sorting by regular sampling, *Journal of Parallel and Distributed Computing 14*, 1992, pp.361-372.

[13] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H. Yamana, S. Sakai, and Y. Yamaguchi, Identifying the Capability of Overlapping Computation with Communication, in *Proc. ACM/IEEE Conf. on Parallel Architectures and Compilation Techniques*, Boston, MA, October 1996, pp. 133-138.

[14] L. G. Valiant, A bridging model for parallel computation, *Communications of the ACM 33*, August 1990, pp.103-111.

[15] T. Welcome, Introduction to the Cray T3E Programming Environment, http://www.nersc.gov/training/T3E/intro9.html.