

## Partitioned Parallel Radix Sort

DongSeung Kim


*Journal of Parallel and Distributed Computing*

### Cite this paper

Downloaded from [Academia.edu](#) 

[Get the citation in MLA, APA, or Chicago styles](#)

### Related papers

[Download a PDF Pack](#) of the best related papers 



[Communication conscious radix sort](#)

Daniel Jimenez-gonzalez

[Sorting using Bltonic network with CUDA](#)

Ranieri Baraglia

[Department of Electrical and Computer Engineering](#)

Nader Bagherzadeh

# Partitioned Parallel Radix Sort \*

Shin-Jae Lee, Minsoo Jeon,  
Dongseung Kim  
Dept. Electrical Engineering  
Korea University  
Seoul, 136-701 Korea  
E-mail: dkim@classic.korea.ac.kr

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

## Abstract

*Partitioned parallel radix sort* is a parallel radix sort that shortens the execution time by modifying the *load balanced radix sort* which is known one of the fastest internal sorts with parallel processing. Parallel sorts usually consist of a few phases of local sort and data movement across processors. In load balanced radix sort, it requires data redistribution in each round for perfect load balancing, whereas in partitioned parallel radix sort, it is needed only once in the first round. The remaining work is only computation and data movement within each processor, requiring no further interprocessor communication. The proposed method has been implemented on IBM SP2, PC Cluster, and CRAY T3E. The experimental results show that partitioned parallel radix sort outperforms the load balanced radix sort in all three machines with various key distributions, by 13% up to 30% in SP2, and 20% to 100% in T3E, 2.5 fold or more in PC Cluster, in the execution time.

**Keywords:** Radix sort, multiprocessor, parallel algorithm, message passing interface, load balancing

## 1 Introduction

Sorting is one of the fundamental and heavily used computation in scientific and engineering applications [9, 10]. Sorting of a certain number of keys has been used in benchmarking

parallel computers or judging the specific algorithm performance when it is experimented on the same parallel machine. Sequential sort often needs  $O(N \log N)$  time if it is based on comparison of keys, and it may not be small if the number of keys gets large. Various parallel sorting algorithms have been devised to speed up the execution such as bitonic sort [1], sample sort [4, 5], column sort [8],  $k$ -sorter [15].

In general, parallel sorts consist of multiple rounds of sequential sort (or often called *local sort*) performed in each processor in parallel, and data exchange among processors to move keys to other processors [7]. The local sort and data exchange may be intermixed and iterated a few times depending on the algorithms used. The time spent in local sort depends on the number of keys. Parallel sort time is the sum of the times of local sort and times for data exchange in all rounds. To make the sort fast, it is important to distribute the keys as evenly as possible throughout the whole rounds of sort, since the execution time is dependent on the most heavily loaded processor in each round [5, 12]. If parallel sort balances its work load perfectly in each round, there will be no further improvement of the time spent in that part. However, the communication time varies depending on the data exchange schemes

---

\*The authors specially thank MHPCC for letting them use the SP-2 machine for the experiments. The work is partially supported by STEPI grant no. 97-NF-03-04-A-01, KRF grant no. 97-E2011, and NSF grant no. INT-9722545.

(e.g. all-to-all, one-to-many, many-to-one), the amount of data and its frequency of communication (e.g. many short messages, or a few long messages), and network topologies (hypercube, mesh, fat-tree) [6, 3]. It was reported that for large number of keys, the communication times occupy a great portion of the sorting time [3, 13]. Our scope of research is to devise a quick parallel sorting method using one of the *conventional* sequential sorting algorithms.

*Load balanced parallel radix sort* [12] (abbreviated from now on as LBR) shortens the execution time by perfectly balancing the load among processors in every round of the sort. *Partitioned parallel radix sort* (abbreviated as PPR) proposed in the paper further improves it by cutting the multiple rounds of data exchange into one. It may have slight load imbalance among processors compared to LBR, nevertheless, the saving in time results in overall gain of performance.

This paper is organized as follows. Section 2 introduces the method and problems of parallel radix sort, section 3 describes and analyzes partitioned parallel radix sort, section 4 gives the experimental results of the sort on parallel machines, and the paper concludes in section 5.

## 2 Parallel Radix Sort

Radix sort is a simple and very efficient sorting method that outperforms many well known comparison-based algorithms. The authors believe that radix sort can be very powerful with efficient implementation. When sort completes, we expect that keys are ordered according to the index of processors  $P_0, P_1, \dots, P_{P-1}$ , besides keys in each processor have also been sorted. Since parallel radix sorts are derived from sequential radix sorts, we first briefly describe them, and ideas of parallel implementa-

tion will be given. Before the detailed description of parallel radix sorts, symbols used in this paper are listed below:

- $P$  is the number of processors.
- $N$  is the problem size, i.e. the total number of keys.
- $n$  is the average number of keys per processor for perfect load balance.
- $g$  is the group of bits used at each round of scanning.
- $r$  is the number of rounds each key goes through. It is the number of rounds of radix sort.
- $b$  is the number of bits of an integer key.
- $M$  is the number of buckets for the selected group of bits. It is determined as  $M = 2^g$ .

Sequential radix sort can be accomplished in two different ways: *radix exchange sort* and *straight radix sort* [11].

**Radix exchange sort** generates and maintains an *ordered queue*. It examines keys one by one. Initially the unordered list of keys is used as an ordered queue. It scans the  $g$  *least significant* bits (in other words,  $g$  rightmost bits) of each key, and places it in a new queue at the position determined by the  $g$  bits. If all keys are examined and placed in the new queue, the round completes. The second round starts by picking up the first key in the ordered queue, scanning the next  $g$  least significant bits, and moving it to another newly generated queue. Then, the second key in the old queue is picked up, scanned, placed, and so on. The remaining keys are treated similarly, and if all keys are moved, the round ends. Keys move back and forth during the round. In the following rounds, the same operations are done as before. After  $r$  rounds (where  $r = \lceil \frac{b}{g} \rceil$ ), all bits are scanned, and the sort is done.

LBR parallelizes the sort on multiple processors by repeating the following process a given

number of times: it builds an ordered queue globally, then divides it into  $P$  *equal sized* segments, allocates them to all processors. In other words, a globally ordered queue is created, then divided equally, and each is assigned to one processor. In the next round a new ordered queue is again generated by  $P$  processors together, then divided equally, and distributed. Load of processors is always perfectly balanced in this scheme.

LBR is reported to outperform fastest parallel sorts by up to 100% in execution time. LBR however requires data redistribution across processors in *every round*, thus, it consumes a considerable amount of time in communication.

**Straight radix sort** does not use an ordered queue, but it initially uses  $M = 2^g$  buckets instead. It picks up keys one by one, examines the  $g$  *most significant bits* (i.e. from left to right, a group of  $g$  bits is selected) of the selected key, places it into the bucket whose index corresponds to the  $g$  bits. Thus, keys with the same  $g$  bits will gather in the same bucket. This process is called *bucket-sort* [11]. When all keys are bucket-sorted, the second round begins, and keys in each bucket are bucket-sorted again using the next  $g$  most significant bits (which are the  $r + 1$ st,  $r + 2$ nd,  $\dots$ , bits to the  $2r$ -th, from the left). It generates  $M$  new subbuckets per bucket in the previous round, thus, there will be  $M^2$  buckets in total. The third and the remaining rounds are executed in the same manner. The sort completes at the round when the bucket sorts consumes the last group of  $g$  bits, which are the most significant  $g$  bits.

In this scheme keys never leave the bucket where they have been placed in a previous round. They are only placed in subbuckets created within it. Data movement is far reduced, but the number of overall buckets explodes eas-

ily, and many buckets may have few keys. Thus it may waste a lot of memory if not carefully implemented.

In parallel implementation, each processor locally scans  $g$  contiguous bits of keys, then bucket-sorts their initial keys using its own  $M$  buckets. Then, only a few disjoint buckets are allocated to each processor. For example, each processor may hold  $M/P$  buckets in average. Now buckets of keys in each processor are sent to the processors in charge of them. Keys in the same buckets from all the processors are collected into a designated processor. It is possible that some processors may be allocated with buckets with a lot of keys while others have few, depending on the characteristics of input keys. Such static/naive partitioning of buckets of keys induces severe load imbalance. PPR solves this problem as described in the next section.

### 3 Partitioned Parallel Radix Sort

Assume that keys are represented as  $v_0, v_1, \dots, v_{N-1}$ , and  $n$  keys are given to each processor in the beginning. We use only  $M = 2^g$  buckets per processor throughout the sort.  $B_{ij}$  represents bucket- $j$  in processor  $P_i$ .

#### 3.1 The algorithm

PPR consists of two phases: *local sort* and *key partitioning*. The former partially sorts keys. After obtaining bucket counts of all processors, and *global-bucket* assignment is made, key movements across processors take place. In fact, this corresponds to the first round of straight radix sort. The latter phase follows sequential radix exchange sort for the keys in the individual processors with no further data movement across processors. The phase consists of  $\lceil \frac{b-g}{g} \rceil$  rounds, thus, PPR needs  $r = \lceil \frac{b}{g} \rceil$  rounds in all.

#### I. Key Partitioning Phase:

Each processor bucket-sorts its keys using the  $g$  most significant bits. From now on, the value of the  $g$  bits of a key is regarded as *the most significant digit (MSD)*. Each key is placed to an appropriate bucket, thus, processor  $P_k$  will send  $v_i$  to bucket  $B_{kj}$ , where  $j = \text{Fg}(v_i)$  and  $\text{Fg}(v)$  is a function that converts the selected  $g$  bits of key  $v$  to the corresponding integer. After the bucket sort, all keys have been sorted internally with respect to the MSD, i.e. the first bucket includes the smallest keys, the second the next smallest,  $\dots$  and the last the largest. Processors now find the numbers of keys (*key counts*) in the buckets they have. They then exchange the key counts together to find a global key distribution map as follows:

For all  $P_0, P_1, \dots, P_{M-1}$ , key counts of  $B_{kj}$  are added up to find  $G_j$ , which is a global count of keys in bucket  $B_{kj}$ s across processors ( $k=0,1, \dots, P-1$ ). Then prefix sums of global key counts are computed. Refer to Figure 1.

Let's consider a hypothetical bucket (called a *global bucket*)  $GB_j$  that includes buckets of keys  $B_{kj}$  from all processors  $P_0, P_1, \dots, P_{P-1}$ . Then  $G_j$  corresponds to the key count of  $GB_j$ . Taking account the prefix sums and the average number of keys ( $n = N/P$ ), global buckets are to be divided into  $P$  groups, each consisting one or more consecutive buckets, so that the total key counts of each group are as equal as possible. The first group will have the first few buckets  $GB_0, GB_1, \dots, GB_{k-1}$  that accomodate approximately  $n$  keys, the second will have buckets  $GB_k, GB_{k+1}, \dots, GB_l$  to accomodate another  $n$  keys, and so on.

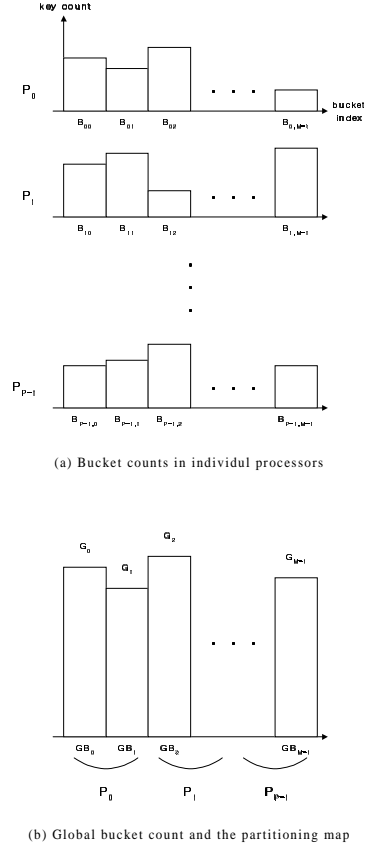


Figure 1: Local and global key count maps and bucket partitioning

The remaining groups will be formed in a similar way. Each group of buckets is now given to one processor, and the processor is the *owner* of the buckets. Notice that the allocation is done in order of processors, i.e. the first group is given to  $P_0$ , the second to  $P_1$ , and so on.

Now all processors send their own buckets of keys to their owners simultaneously. A bucket of keys in a processor either stays there if the processor is the owner, or is sent out to its owner. When the global key movement finishes, keys have been sorted partially across processors, since buckets of keys have been placed in the order of  $P_0, P_1, \dots, P_{P-1}$  and any key in  $GB_i$  is smaller than any key in  $GB_j$  if  $i < j$ . The remaining work is to sort keys independently in each processor.

## II. Local Sort Phase:

Keys in each processor are now sorted locally by all processors, to make all  $N$  keys in order. Sequential radix exchange sort is performed initially with a group of right-most  $g$  bits, then with a group of the next  $g$  bits,  $\dots$ , until all  $b - g$  bits have been used. Only  $b - g$  bits are examined because keys have been sorted in Phase I with respect to the left most  $g$  bits. Phase II needs  $\lceil (b - g)/g \rceil$  rounds.

The success of PPR relies on how evenly the keys are distributed at the first phase. It is not very likely that each processor has exactly the same number of keys. We can allocate in Phase I equal number of keys if we would divide some buckets into pieces and they were given to different processors, but it needs additional computation later since in such a case, local sort of Phase II will not produce the overall sorted output. This is because keys having the same MSD stay in different processors and are

sorted individually. Unless they are rearranged by merging, a correct sorted sequence can not be obtained. Thus, we avoid dividing a bucket further, and keys are distributed to processors *bucket by bucket*.

PPR resembles *sample sort* [4, 5] in the aspect of data partitioning and local sort. In sample sort, after keys have moved according to the *splitters* to each processor, they are partially ordered with respect to the index of processors, thus further movement of keys across processors is not needed. One thing different is that in sample sort the global key distribution is not known until keys actually move to designated processors, while in partitioned radix sort, it is known before the costly data movement. Thus, it is possible to *adjust* the value  $g$  before the actual key move. If current partitioning is judged not to give good balance in work load,  $g$  should be enlarged so that keys are spreaded out further into many new buckets and grouping of buckets can be better controlled. For example, if  $g$  is increased by 2 (bits), the number of buckets grows four fold. The process can be repeated until a satisfactory result is obtained.

Choosing a greater  $g$ , however, increases the frequency of data exchange since we move keys bucket by bucket. Hence it could lengthen the communication time. In that case, we move keys based on *coarse-grained* buckets produced by the smaller  $g$ , except we use the *fine-grained* ones at the partition boundaries, i.e. the last bucket in one processor and the first bucket in the next processor. The scheme is adopted in our experiments.

## 3.2 Performance analysis

Since the previous work of LBR has included comparisons with other competitive sorts, we will use only LBR for performance comparison with PPR. Both PPR and LBR will be executed

on the same machine.

The execution time of LBR should reflect  $r = \lceil b/g \rceil$  iterations of local bucket-sort, one transpose of key counts, and a set of key send/receive operations. The parallel time of PPR consists of three terms: the times for  $r$  rounds of local bucket-sort, for one transpose of key counts, and for one round of key movement.  $T_{\text{LBR}}$  and  $T_{\text{PPR}}$ , respectively, can be expressed as

$$T_{\text{LBR}}(N, P) = r T_{\text{seq}}\left(\frac{N}{P}\right) + r T_{\text{tp}}(M, P) + \sum_{i=1}^r T_{\text{move}}(D_i, P) \quad (1)$$

$$T_{\text{PPR}}(N, P) = \sum_{j=1}^r T_{\text{seq}}\left(\frac{N}{P}(1.0 + \Delta_j)\right) + T_{\text{tp}}(M, P) + T_{\text{move}}(D'_1, P) \quad (2)$$

where  $M = 2^g$ ,  $T_{\text{seq}}(n)$  is the time for sequential radix sort of  $n$  keys in a processor,  $T_{\text{tp}}(M, P)$  is the time for transposing  $M$  key counts of bucket per processor,  $D_i, D'_j$  are the amounts of data to move across processors during redistribution at round  $i, j$  for LBR and PPR respectively,  $T_{\text{move}}(D_i, P)$  is the time for exchanging  $D_i$  keys per processor on  $P$  processors, and  $\Delta_j$  represents maximum positive deviation from the perfect balance at the  $j$ -th round, accounting for the degree of load imbalance. Here we assume that all processors are equally powerful, and have the same communication capability.

A performance measure (speed up), denoted as  $\eta$ , of PPR over LBR is defined as the ratio of  $T_{\text{LBR}}$  to  $T_{\text{PPR}}$ . If any improvement is obtained, the measure should be greater than 1.0.

$$\eta = \frac{V(N, M, P)}{W(N, M, P)} \quad (3)$$

$$V(N, M, P) = r T_{\text{seq}}(n) + r T_{\text{tp}}(M, P)$$

$$W(N, M, P) = \sum_{j=1}^r T_{\text{seq}}(n(1.0 + \Delta_j)) + T_{\text{tp}}(M, P) + T_{\text{move}}(D_1, P)$$

where  $n = \frac{N}{P}$  is used.

Let's refine the equation (3) under some assumptions. Suppose values of input keys are evenly distributed throughout the range, for example, 0 to  $2^b - 1$  for positive keys (*uniform* initialization introduced in the next section generates keys like this). Communication time for key movement is minimum if keys are equally divided and sent to every processor, other than a great portion is given to a few processors. In such a case, the last terms in equations (1) & (2) can be expressed as  $r T_{\text{move}}(D_E, P)$  where  $D_E = n \frac{P-1}{P}$ . Now  $\eta$  is

$$\eta = \frac{V(N, M, P)}{W(N, M, P)} \quad (4)$$

$$\begin{aligned} V(N, M, P) &= r T_{\text{seq}}(n) + r T_{\text{tp}}(M, P) + r T_{\text{move}}(D_E, P) \\ W(N, M, P) &= \sum_{j=1}^r T_{\text{seq}}(n(1.0 + \Delta_j)) + T_{\text{tp}}(M, P) + T_{\text{move}}(D_E, P) \end{aligned}$$

If PPR is able to keep load imbalance minimum so that  $\Delta_j$  can be negligible, the first terms in nominator and denominator of equation (4) are nearly equal. Substituting the value  $\Delta_j = 0$  and dividing both nominator and denominator by  $r T_{\text{seq}}(n)$  yields the following relationship:

$$\begin{aligned} \eta &\approx \frac{1 + \frac{T_{\text{tp}}(M, P) + T_{\text{move}}(D_E, P)}{T_{\text{seq}}(n)}}{1 + \frac{1}{r} \cdot \frac{T_{\text{tp}}(M, P) + T_{\text{move}}(D_E, P)}{T_{\text{seq}}(n)}} \\ &= \frac{1 + F(n, M, P)}{1 + \frac{1}{r} \cdot F(n, M, P)} \end{aligned} \quad (5)$$

where  $F(n, M, P) = \frac{T_{\text{tp}}(M, P) + T_{\text{move}}(D_E, P)}{T_{\text{seq}}(n)}$ .  $\eta$  is greater than 1.0 as long as  $F$  is positive, and is an increasing function which grows asymptotically to  $r$ . Notice that  $T_{\text{seq}}(n)$  is not a function of communication speed of the machine. If

the communication speed is low, the nominator of  $F$  is large, so is  $F$ , then  $\eta$  is enlarged. In other words, the improvement gets greater as the ratio of communication cost to the overall execution time is increased.

Consider the case like *uniform* such that after bucket sort in Phase I keys are evenly placed in all buckets, thus only  $\frac{P-1}{P}$  of keys in each processor will be sent out to their owners. In such a case,  $\eta$  becomes the value given in (5). Thus, for sorting of keys having randomly distributed values in its whole range, PPR will have great improvement in parallel computers having slow communication.

Although we have assumed a certain key characteristics above, for most of sort instances the algorithm balances the work load reasonably well, thus, it performs satisfactorily. Observe the experimental results below that support the fact.

## 4 Experiments and Discussion

Partitioned parallel radix sort is experimented on three different parallel machines: IBM SP2, Cray T3E, and PC cluster. PC cluster is a set of 16 Pentium-II personal computers interconnected by a fast ethernet switch. T3E is the fastest machine among them, as long as the computational speed is concerned. However, in terms of communication speed (bandwidth) SP2 is the fastest, T3E the next, and PC cluster the slowest. This fact is important when we compare the difference of improvement among the three in the experiments.

For inputs of sort,  $N/P$  keys are synthetically generated in each processor using some random number generation schemes, called *uniform*, *gauss*, *stagger* [12]. Uniform is supposed to create keys with uniform distribution. Gauss forms keys with a certain Gaussian distribution. Stagger produces specially distributed keys as described in [4]. We run the programs on 2, 4,

..., 16, up to 64 processors, each with 1M, 2M, ..., 64M keys. Keys of sorts are 32-bit integers for SP2 and PC cluster, and 64-bit integers for T3E. For the portability of code, we use C with MPI communication library [14]. Practically the complexity of the code compared with LBR is slightly increased, but the code is quite the same except the first phase. To avoid the effect of sharing of CPU with other users, multiple runs with batch processing are performed, then, the best results are selected which are thought to be least interfered by others due to sharing of the interprocessor communication network.

Among many experiments we have performed, only a few representative results are shown here. In the graphs, SJ (drawn in solid lines) and LB (broken lines) represent the experimental results of PPR and LBR, respectively.

We first verify that PPR can reduce the communication time while it tolerates load imbalance. We expect the communication time be cut down to 1/4 and 1/8 at maximum with  $g = 8$ , compared to LBR for sorts of 32-bit and 64-bit integer keys, respectively. As seen in Figures 2-4, there is a great reduction of communication times: they are now about 1/4 for 32-bit keys in SP2, and around 1/6 for 64 bit integers in T3E.

Except the initialization of uniform, balancing the work load is not easily obtained, especially if the keys are gathered in a few buckets. Gauss is the hardest, and the corresponding load distribution is shown in Figure 5. It shows the maximum of 5.2% of load imbalance against the perfect balanced case. In all other experiments, it is less than 6%, which tells that the load imbalance is not so severe as to significantly impair the overall performance of PPR.

Next, the overall speedup  $\eta$  is investigated.



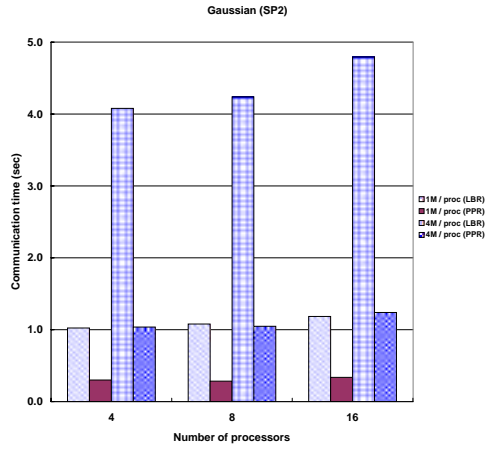


Figure 2: Comparison of communication times of PPR and LBR on SP2 (gauss initialization is used)

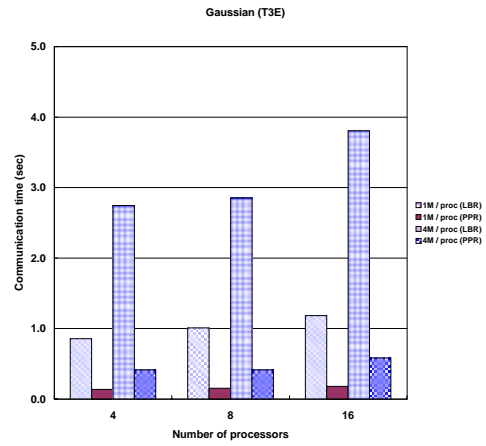


Figure 4: Comparison of communication times on T3E (gauss)

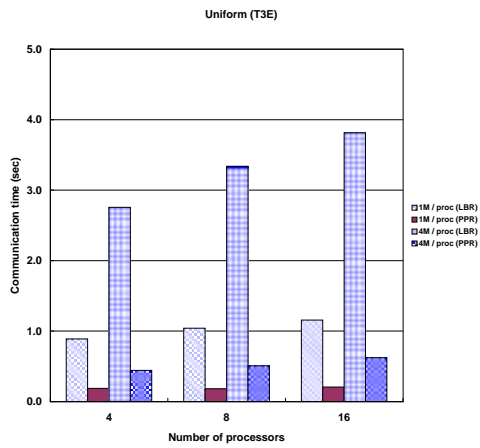


Figure 3: Comparison of communication times on T3E (uniform)

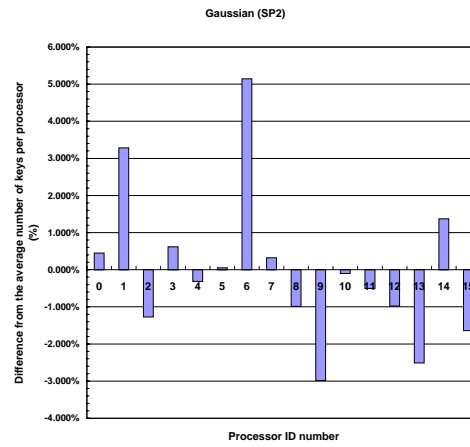


Figure 5: Percentage deviation of work load from perfect balance on SP2 (gauss)

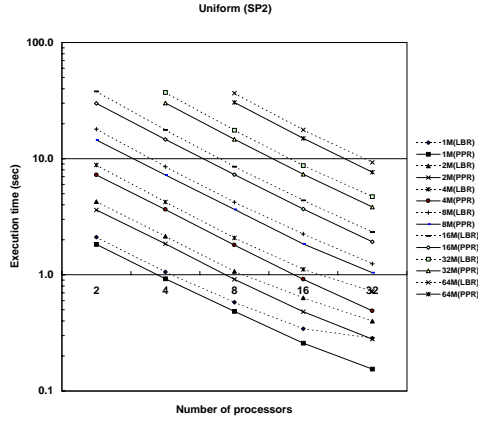


Figure 6: Execution times on SP2 (uniform)

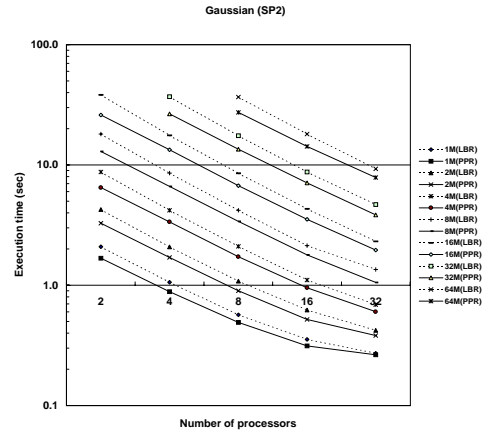


Figure 7: Execution times on SP2 (gauss)

If we use the same  $g$  regardless of key characteristics, *gauss* results in more concentration of keys in a few bucket than *uniform*. Keys having uneven bucket counts are harder for PPR to balance. By using the refining scheme of  $g$ , the performance of PPR does not fall much, as seen in Figures 6 & 7, and Figures 9 & 10. We have increased  $g$  to 10, but bucket movement are based on  $g = 8$  except at the boundaries, as mentioned before.

When the number of processors gets bigger, the number of buckets allotted per processor becomes smaller. This may easily lead to uneven distribution of keys if keys have values in a small range. It can also be solved by enlarging the number of buckets. The experimental results can be observed at the bottom right corner of Figures 6–10.

According to the analysis given in section 3.2, SP2 is expected to yield the least improvement among the three, since the communication speed relative to its own computation speed is the fastest. We have found the fact that in T3E the communication portion in sorting time is greater than SP2. In addition, since sorting inputs are 64-bit integers, more improvement of PPR over LBR is expected due to larger  $r$ .

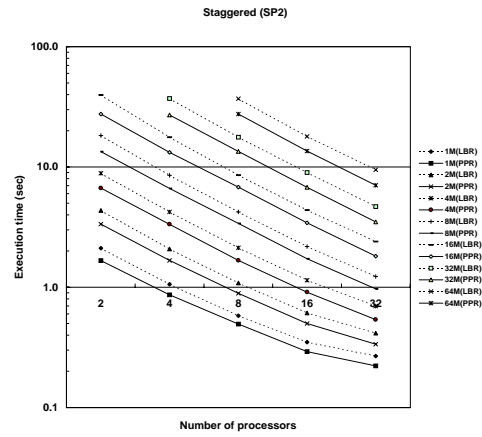


Figure 8: Execution times on SP2 (stagger)

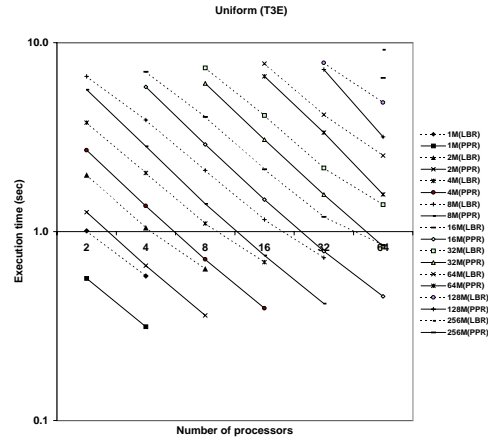


Figure 9: Execution times on T3E (uniform)

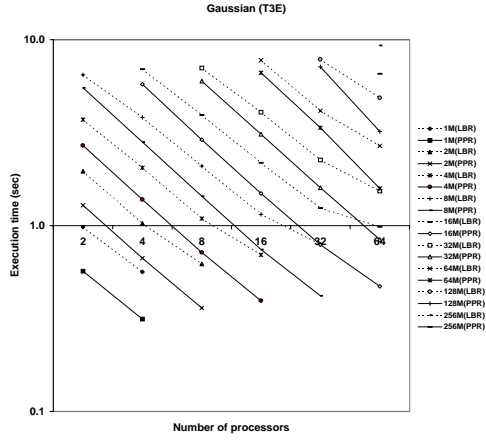


Figure 10: Execution times on T3E (gauss)

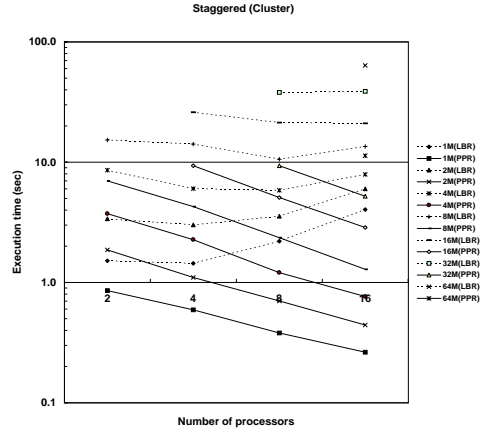


Figure 12: Execution times on PC cluster (stagger)

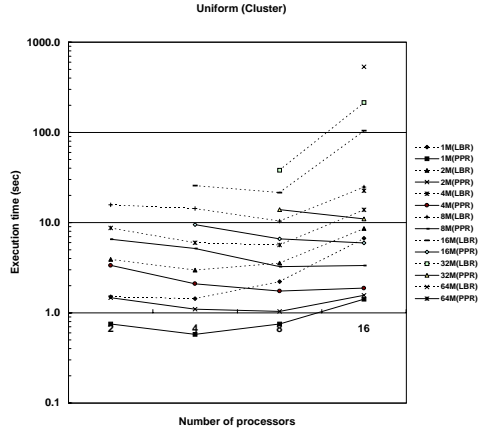


Figure 11: Execution times on PC cluster (uniform)

More enhancement on T3E can be observed in Figures 9 & 10 compared to Figures 6 & 7, respectively.

In PC cluster, the network is so slow that the two parallel sorts are *slower* than the uniprocessor sort for the cases of  $P \geq 8$  as shown in Figures 11-12. Nevertheless, for all cases PPR has remarkable performance over LBR since the communication times dominate the computation times. Table 1 lists the performance.

## 5 Conclusion

We have proposed partitioned parallel radix sort. It partially sorts and divides the overall keys to processors in the first phase so that each processor has nearly equal number of keys, then does sequential radix sort individually for the assigned keys in the second phase. It improves the sort performance by cutting a great portion of communication time. The reduction comes from the allocation of keys to processors in such a way that processors no longer require redistribution of keys to other processors after the first phase. The previous parallel radix sort (called load balanced parallel radix sort) [12] needs key movement at every round.

It has been observed that partitioned parallel radix sort always performs better than the previous scheme regardless of data size, the number of processors, and key initialization scheme. The enhancement is greater on the machines that have slower communication with respect to the speed of its own computation. The fact is important because although the absolute speed of interprocessor communication has been improved much in recent distributed-memory parallel computers, communication operations are

Table 1: Execution time of sorts on 4-processor PC cluster

Keys	LBR	PPR	$\eta$
uniform			
1M	1.434	0.577	2.485
2M	2.967	1.094	2.712
4M	5.974	2.090	2.858
8M	14.30	5.166	2.768
16M	25.660	9.480	2.706
gauss			
1M	1.483	0.619	2.396
2M	3.001	1.086	2.763
4M	5.932	2.147	2.763
8M	14.209	4.530	3.137
16M	26.061	8.799	2.962
stagger			
1M	1.442	0.593	2.432
2M	3.008	1.101	2.732
4M	6.034	2.268	2.660
8M	14.129	4.279	3.302
16M	26.007	9.342	2.784

still regarded costly and time-consuming compared to computation operations.

To summarize the experimental results, on SP-2, the improvement of the sorting time is 13% to up to 40%, for T3E 20% to 100% improvement is obtained, and PC cluster gives over 2.5 fold speedup.

## References

- [1] M. E. Batcher, Sorting Networks and their applications, *Proceedings of AFIPS Conference*, pp. 307-314, 1968.
- [2] R. Beigel and J. Gill, Sorting  $n$  objects with  $k$ -sorter, *IEEE Transactions on Computers*, vol. C-39, pp. 714-716, 1990.
- [3] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R.P. Martin, Fast parallel sorting under LogP: experience with the CM-5, *IEEE Trans. Computers*, Vol. 7(8), Aug. 1996.
- [4] D. R. Helman, D. A. Bader, and J. JaJa, Parallel algorithms for personalized communication and sorting with an experimental study, *Procs. ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, pp. 211-220, June 1996.
- [5] J. S. Huang and Y. C. Chow, Parallel Sorting and Data Partitioning by Sampling, *Procs. the 7th Computer Software and Applications Conference*, pp. 627-631, November 1983.
- [6] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.
- [7] J. JaJa, *Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
- [8] F. T. Leighton, Tight Bounds on the Complexity of Parallel Sorting, *IEEE Transactions on Computers*, C-34: pp. 344-354, 1985.
- [9] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Addison-Wesley, Morgan Kauffman, 1992.
- [10] W. A. Martin, Sorting, *ACM Computing Surveys*, Vol. 3(4), p.p. 147-174, 1971.
- [11] Sedgewick, *Algorithms*, Wiley, 1990.
- [12] A. Sohn and Y. Kodama, Load balanced parallel radix sort, *Procs. 12th ACM Int'l Conf. Super computing*, Melbourne, Australia, July 14-17, 1998.
- [13] A. Sohn, Y. Kodama, M. Sato, H. Sakane, H. Yamada, S. Sakai, Y. Yamaguchi, Identifying the capability of overlapping computation with communication,

*Procs. ACM/IEEE Parallel Architecture and Compilation Techniques*, Boston, MA, Oct. 1996.

- [14] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, June 1995.
- [15] S. Q. Zheng, Algorithm for sorting arbitrary input using a fixed-size parallel sorting device, *Procs. Int'l Conf. Parallel Processing*, vol. A, p.p. 95-99, 1996.