Integer Sorting on Shared-Memory Vector Parallel Computers

Kenji Suehiro
NEC Corporation
4-1-1 Miyazaki, Miyamae-ku
Kawasaki, 216-8555 Japan
+81-44-856-2183
suehiro@ccm.cl.nec.co.jp

Hitoshi Murai
NEC Corporation
4-1-1 Miyazaki, Miyamae-ku
Kawasaki, 216-8555 Japan
+81-44-856-2183
murai@ccm.cl.nec.co.jp

Yoshiki Seo
NEC Corporation
4-1-1 Miyazaki, Miyamae-ku
Kawasaki, 216-8555 Japan
+81-44-856-2183
seo@ccm.cl.nec.co.jp

ABSTRACT

This paper describes new fast integer sorting methods for single vector and shared-memory parallel vector computers, based on the bucket sort algorithm. Existing vectorization methods for bucket sort have made great efforts to avoid store conflicts of vector scatter operations, and therefore are not so efficient. The vectorization methods shown in this paper—the retry method, the split vector method and the mask vector method—all actively utilize the nature of the store conflicts to achieve high performance. The parallelization method in this paper uses a feature of shared-memory machines and dynamically changes the partitioning of histogram arrays without any overhead. By combining the retry and the parallelization methods, we got the world's fastest results for the IS program (Class B) in the NAS Parallel Benchmarks on the NEC SX-4. Our methods are also applicable to a wide range of particle simulation programs.

Keywords

Integer sorting, particle pusher, store conflict, vectorization, parallelization.

1. INTRODUCTION

There is a class of programs called "particle pusher" codes, which accumulate data into target arrays using indirect index vectors. These codes often appear in application programs of high performance computing, and therefore are very important. The NAS Parallel Benchmarks (NPB) include integer sorting (IS) as an example of "particle pusher" codes.

We have developed new fast integer sorting methods for single vector computers and shared-memory parallel vector computers. Using these methods, we got the world's fastest results for the NPB IS on the NEC SX-4. Our methods are also applicable to a wide range of particle simulations.

ICS 98 Melbourne Australia

Bucket sort, a basic sorting algorithm which we use, consists of the following three phases:

(a) Compute a histogram of keys to sort.

(b) Compute running sum of the histogram.

(c) Rank keys by the running sum.

Better vectorization of phase (a) is the key to faster sorting, since good vectorization methods for phase (b) and (c) are known. Phase (a) is also a variation of "particle pusher" codes, therefore if we can vectorize phase (a) efficiently, we could improve vector performance of particle simulation programs using the same techniques. This paper focuses on the vectorization of phase (a). We vectorize phase (b) using vector reduction instructions and phase (c) using the method of Ishiura et al.[3]

In Section 2, we define the problem to solve and clarify the background assumptions. In Section 3, we present the retry method, a vectorization method we developed for vector computers. In Section 4, we discuss two other vectorization methods we developed and tried. In Section 5, we present the parallelization method we developed for shared-memory parallel vector computers. In Section 6, we show some evaluation results on the SX-4. In Section 7, concluding remarks and future plans are presented.

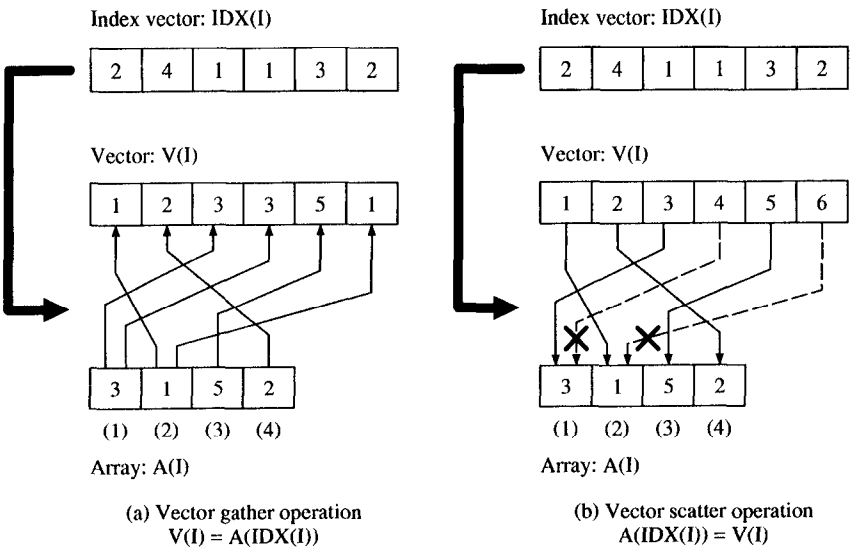
2. PRELIMINARIES

2.1 Definition of Integer Sorting

This paper uses the same definition of "integer sorting" as that in the NPB document[1]:

A sequence of keys *key(i)*, where $i = 0, ..., M-1$, will be *sorted* if it is arranged in non-descending order, i.e. $key(i) \le key(i+1)$ for all i. The *rank* of a particular key is the index value i that the key would have if the sequence of keys were sorted. *Ranking* is the process to obtain ranks for all the keys. *Sorting* is the process to permute the keys to produce a sorted sequence. Sorting is not required to be stable; equal keys do not need to retain their original order.

This paper discusses vectorization and parallelization of ranking.



(a) Vector gather operation
V(I) = A(IDX(I))

(b) Vector scatter operation
A(IDX(I)) = V(I)

Figure 1: Vector Gather/Scatter Operation

2.2 Assumption on Vector Gather/Scatter Operations

This paper assumes the following characteristics of a vector gather/scatter operation:

(A) A processor correctly loads all vector elements by any index vector in a vector gather operation, even if there are two or more equal index values in the index vector. For example, Figure 1-(a) shows that both *V(1)* and *V(6)* are loaded correctly although their indices are the same.

(B) A processor correctly stores exactly one vector element of those which should be stored to the same location by a vector scatter operation. That is, when there are two or more equal index values in the index vector, one of the corresponding elements "survives" and is guaranteed to be stored. For example, Figure 1-(b) shows that *V(3)* is stored correctly and *V(4)* is not.

(C) A processor always stores the same elements correctly for the same index vector in a vector scatter operation. That is, the set of "surviving" elements is always the same combination for a given index vector. For example, the processor illustrated in Figure 1-(b) always stores *V(3)* to *A(1)* when the index vector *IDX* is *{2, 4, 1, 1, 3, 2}*.

Most existing vector processors (including the SX-4) conform all the above assumptions.

3. RETRY METHOD

3.1 Overview

The retry method is a vectorization method we developed for single vector computers. It vectorizes Phase (a) of bucket sort. As mentioned in Section 1, the method is directly applicable to "particle pusher" codes and can improve the performance of particle simulation programs.

The essence of Phase (a) is to increment histogram *hist(k)* for each *{k | k = key(i), i = 0, ..., M-1}*. More precisely, a processor loads *hist(k)*, increments it, and stores it back to *hist(k)* for each *k = key(i)*. If we vectorize the process straightforwardly, the loads will become vector gather operations and the stores will become vector scatter operations with index vectors of *k = key(i)*, which may cause store conflicts. Existing vectorization methods usually try to avoid these conflicts, while our retry method accepts and even uses the conflicts to achieve high performance.

The main merits of the retry method are the following. First, it needs only a small amount of extra memory for vectorization. Secondly, it needs only a small amount of extra computation for vectorization. Finally, it is fully vectorizable and therefore it can be fast.

3.2 Procedure

The retry method breaks Phase (a) of the bucket sort algorithm down into the following four operations:

(1) Compute a histogram with a key vector, ignoring store conflicts.

(2) Detect store conflicts for the key vector and gather those key elements causing store conflicts (called *conflicted keys*) into a *retry queue*, which saves conflicted keys for later retry. The computation results of Step (1) for conflicted keys are not stored properly at this point. Later we describe a technique to detect the store conflicts.

(3) Repeat Step (1) and (2) for all keys.

(4) Repeat Loop (3) for all keys in the retry queue, until no key remains.

Figure 2 illustrates an example of the procedure. Twelve keys are processed by 4-way vector execution (shown in the upper left blocks). In Step (1), the first three elements of the first key vector properly counted into the histogram array, while the fourth is not because of a conflict with the third key. Therefore in Step (2), the fourth element is gathered into the retry queue by a vector gather operation (denoted by an arrow to the upper middle blocks). In Loop (3), the process so far is repeated for the remaining key vectors. At this point, nine of all twelve keys are

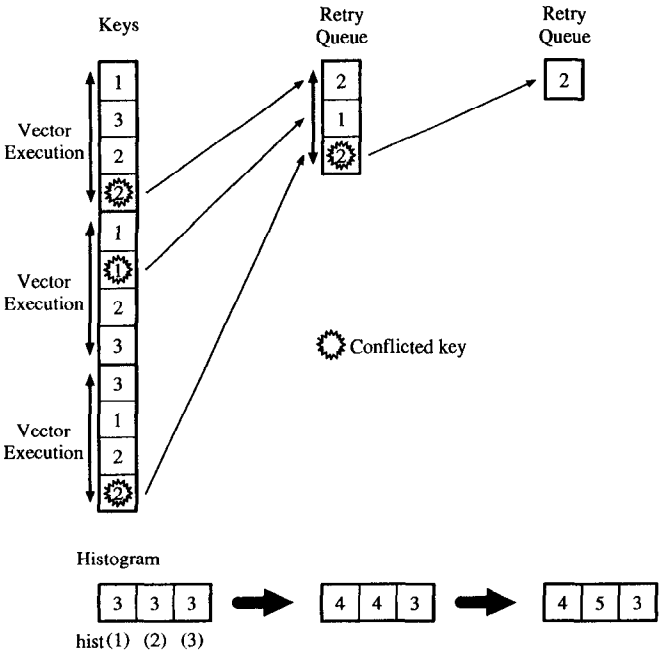


Figure 2: Retry Method

counted into the histogram (the lower left blocks), and the other three keys are gathered into the retry queue. Then in Loop (4), the process so far is repeated for keys in the retry queue until no key remains. The first two keys in the retry queue are counted by the first retry (the lower middle blocks) and the last is stored again to the queue (the upper right blocks), which is finally counted by the second retry (the lower right blocks).

3.3 Detecting Store Conflicts

We use a very simple method to detect store conflicts in Step (2). We prepare a work array of the same shape as the histogram array. Let K be a key vector of length V to be examined. A processor stores the trivial sequence $T = \{1, ..., V\}$ to the work array with the index vector K, then loads it back immediately with the same index vector K. If there is no store conflict, the loaded vector T' will be exactly the same as T. If there is a conflict, suppose the i-th and j-th elements of K are equal, then the i-th element of T' will have the value j (or vice versa) since only one store of the two is done correctly when storing T to the work array. So the processor can detect conflicts by comparing T and T' element-wise; it can get a mask vector by vector comparison. This mask represents which elements of the key vector should be recomputed, and can be used for gathering conflicted keys in vector gather operations.

Figure 3 illustrates an example of the store conflict detection. The key vector to be examined is $K = \{2, 4, 1, 1, 3, 2\}$ (shown in the upper left blocks). At first, a processor stores a trivial sequence $T = \{1, 2, 3, 4, 5, 6\}$ to the work array with the index vector K (the lower left diagram). Because K has two 1's at its third and fourth elements, one of the corresponding elements of T cannot be stored correctly. (Here the fourth cannot.) Similarly, the sixth element of T cannot be stored by the conflict of two 2's in K. Then, the processor loads T' back from the work array with the same index vector K (the lower right diagram) and compares T' with T (the upper right blocks). The difference of the fourth and sixth elements shows that, with the index vector K, the stores of the fourth and sixth elements always fail due to conflicts.

3.4 Considerations

If no conflict occurs, all keys in a key vector are processed without retry. Otherwise, Assumption (A) and (B) in Section 2 guarantees that one key is processed for each set of equal keys. That means at least one key per a key vector is processed and therefore the repetition in Loop (4) always terminates. Assumption (C) guarantees that the conflict pattern detected in Step (2) are the exactly same as that occurs in Step (1).

The retry method is fully vectorizable and therefore can be performed fast. It needs some extra computation for store

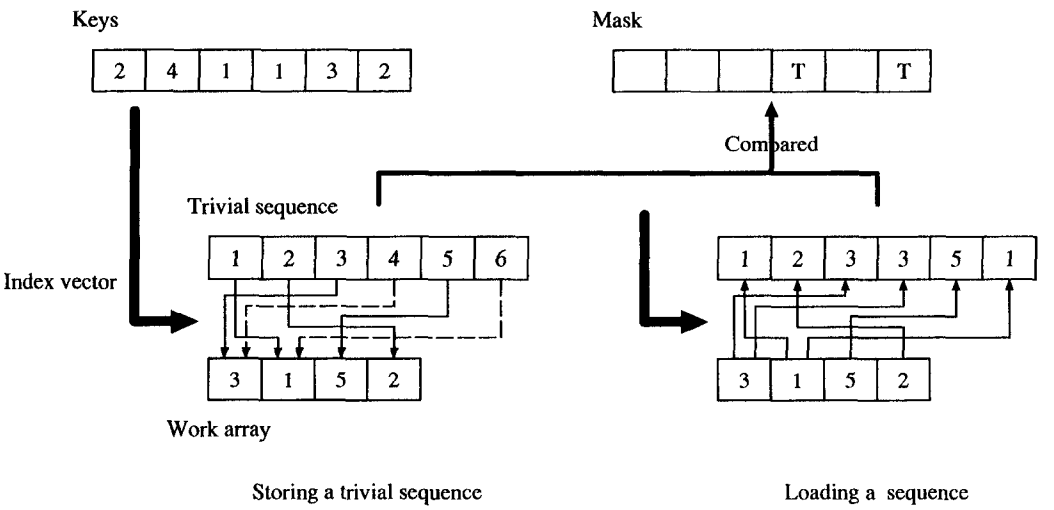


Figure 3: Detection of Store Conflicts

conflict detection and for recomputation of conflicted keys. The amount of computation for the conflict detection is three vector operations (a load, a store, and a compare) per key vector and so is very little. The amount of computation for the retry completely depends on input data, but is usually not so much.

The retry method needs some extra memory for the work array and for the retry queue. As for the work array, it actually needs no extra memory on implementation because the histogram array itself can be used for this purpose by performing Step (1) and (2) simultaneously. That is, a processor at first loads current histogram values from the histogram array to its vector register R_1, and increments it. Then the processor stores a trivial sequence preloaded on register R_2 to the histogram array, loads it back to register R_3, and compares R_2 and R_3 to detect conflicts. Finally it stores the new histogram values kept on R_1 to the histogram array. This works correctly because all loads and stores above use an identical index vector.

As for the retry queue, it requires an array of almost the same size as input keys in the worst cases. That can be a problem when there are not enough memory. We can reduce that memory size by modifying the retry method procedure so that the recomputation in Loop (4) is done whenever there are enough conflicting keys in the queue to form a vector. For example:

(1') Compute a histogram with a key vector, ignoring store conflicts.

(2') Detect store conflicts for the key vector and gather conflicted keys into a retry queue.

(3') If there are enough elements to form a vector in the retry queue, get a key vector from them and do Step (1') and Step (2') for it.

(4') Repeat Step (1') through (3') for all keys.

(5') Repeat Loop (4') for remaining keys in the retry queue, until no key remains.

In this procedure, the retry queue requires only as much memory as two index vectors. (More precisely, the maximum number of keys in the retry queue is *(2V-2)*, where *V* is the vector length. That occurs when Step (2') gathers *(V-1)* conflicted keys to the retry queue already having *(V-1)* elements. Step (3') makes sure that the queue have at most *(V-1)* keys before Step (2') starts, and as mentioned above, Step (2') can gather at most *(V-1)* conflicted keys.) The performance of the modified procedure may be a little worse than the original, because the process in Step (3') such as decision and queue manipulation needs some extra computation.

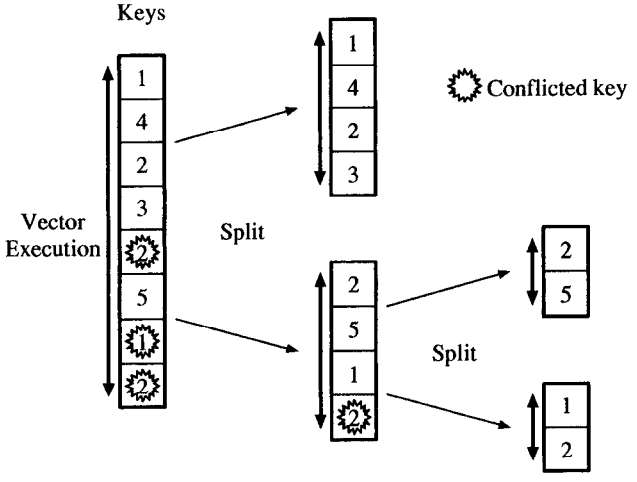


Figure 4: Split Vector Method

4. OTHER VECTORIZATION METHODS

We developed and tried two other vectorization methods as well, called the split vector method and the mask vector method. This section describes them briefly.

4.1 Split Vector Method

The split vector method detects store conflicts before computation, and repeatedly splits key vectors until the conflicts are resolved.

(1) Detect store conflicts for a key vector by the same way as retry method.

(2) If no conflict exists, then compute a histogram with the key vector. Otherwise, split the key vector into two half vectors and apply the procedure so far to both of them recursively.

(3) Repeat Step (1) and (2) for all keys.

Figure 4 illustrates an example of the procedure. In Step (1), a key vector of eight keys $K = \{1, 4, 2, 3, 2, 5, 1, 2\}$ is examined (shown in the left blocks). Because the fifth, seventh and eighth key cause store conflicts, Step (2) splits the vector K into two half vectors $K_1 = \{1, 4, 2, 3\}$ and $K_2 = \{2, 5, 1, 2\}$, and both are examined again (the middle blocks). The first half K_1 does not have conflicts, so a histogram can be computed directly from the key vector (the upper middle blocks). The second half K_2 still has conflicts (the lower middle blocks), so it is split again into $K_{21} = \{2, 5\}$ and $K_{22} = \{1, 2\}$ (the right blocks). They do not have conflicts anymore, so a histogram is calculated from them.

An advantage of this method is that it needs almost no extra memory for vectorization. A disadvantage of this method is that the vector length of computation tends to be short and therefore vector performance may be worse than the processor's peak performance.

4.2 Mask Vector Method

The mask vector method immediately retries the computation for conflicting keys using a mask vector, instead of gathering them into a retry queue for later recomputation.

(1) Set a mask to all-true.

(2) Compute a histogram with a key vector where the mask elements are true.

(3) Detect store conflicts for the key vector where the mask elements are true by the same way as retry method. Set mask elements corresponding to the non-conflicted keys to false.

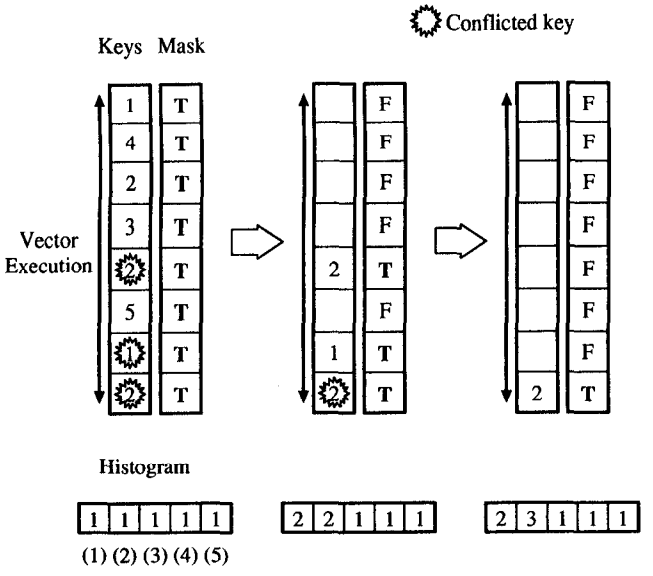


Figure 5: Mask Vector Method

(4) Repeat Step (2) and (3) until the mask becomes all-false.

(5) Repeat Step (1) through (4) for all keys.

Figure 5 illustrates an example of the procedure. The key vector to be processed is $K = \{1, 4, 2, 3, 2, 5, 1, 2\}$. In the first try, Step (2) computes a histogram with the all-true mask set by Step (1) and the histogram array H becomes $\{1, 1, 1, 1, 1\}$ counting the first, second, third, fourth and sixth keys in the key vector. The fifth, seventh and eighth are not counted due to conflicts, which are detected in Step (3). The mask elements corresponding to the conflicted keys are kept true and the others are turned to false. In the second try, Step (2) computes a histogram with the mask $M = \{F, F, F, F, T, F, T, T\}$ and the histogram H becomes $\{2, 2, 1, 1, 1\}$ counting the fifth and seventh keys. Again the eighth key are not counted due to conflicts, and the mask M becomes $\{F, F, F, F, F, F, F, T\}$ in Step (3). Then in the third try, the eighth key is counted at last and H becomes $\{2, 3, 1, 1, 1\}$. The mask M becomes all-false and therefore the repetition terminates.

An advantage of this method is that it needs almost no extra memory for vectorization. A disadvantage of this method is that vector operations with sparse masks may degrade performance.

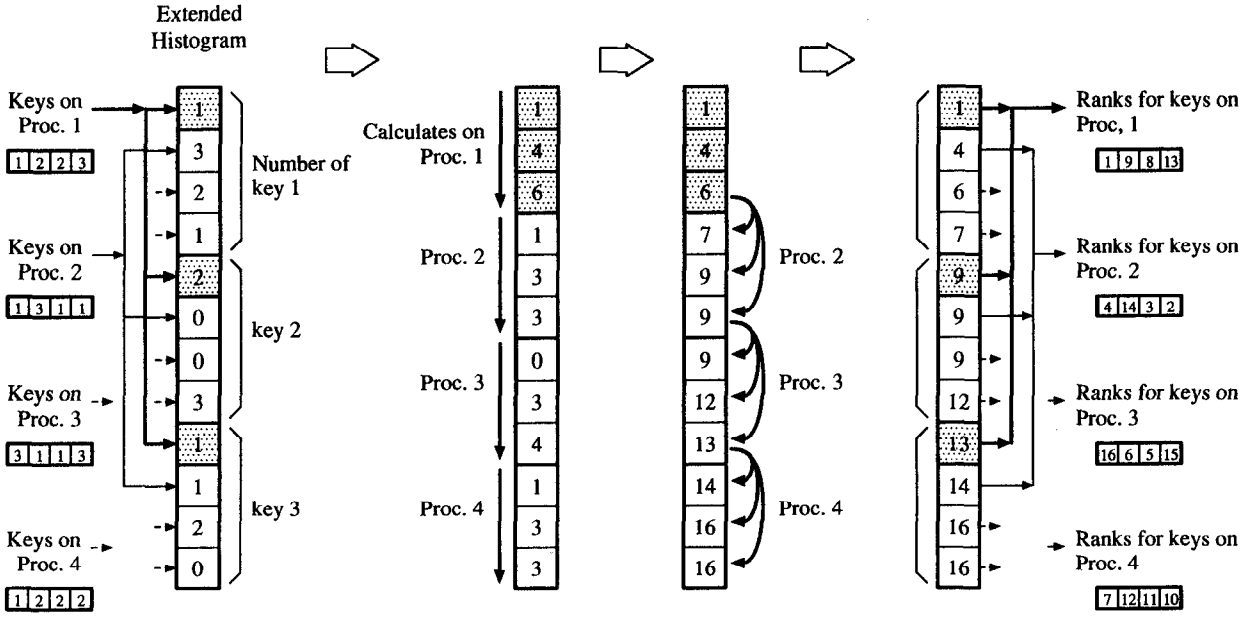
5. PARALLELIZATION OF BUCKET SORT

This section describes the parallelization method we developed for shared-memory vector parallel computers. The method uses a merit of shared-memory machines to dynamically change partitioning and assignment of arrays to processors without any cost. This method is fully parallelized and therefore efficient.

5.1 Procedure

Using our method, bucket sort is parallelized as follows:

(0) Partition the input keys into P equal sets and assign them to P processors. Create P copies of the histogram array. Because the order of equal keys is arbitrary, here we decide to give a smaller rank to a key on a smaller-



(a) Compute histograms (b) Calculate running sums (c) Offset running sums (d) Rank keys

Figure 6: Parallelization of Bucket Sort

numbered processor.

(1) Compute a histogram on each processor with its own keys by some vectorization method such as retry method. Processor p ($p = 0, ..., P-1$) exclusively uses the part of the histogram array such that an index i to the array conforms mod (i, P) = p. In other words, the histogram array is distributed by the "cyclic" format to processors. The "smaller rank for smaller processor" rule is naturally realized by using the histogram array in that way.

(2) Compute running sum of the whole histogram. Change the distribution of the histogram array to the "block" format, compute local running sum on each processor, and offset it by total sums of preceding processors. Note that the change of array distribution and the reference to other processors' sums are done without any cost by using shared-memory.

(3) Rank keys by the running sum on each processor. Change the distribution of the histogram array back to "cyclic", and rank keys with it on each processor. Again, the distribution changes with no cost.

Figure 6 illustrates an example of the procedure. There are 16 keys to sort, distributed to four processors equally. The histogram array H is extended to four times and initially distributed to the four processors in the "cyclic" format. In the figure, the hatching blocks denote the elements assigned to Processor 1. In Step (1), every processor computes its own histogram (shown in figure (a)). In this step, Processor 1 uses *H(1)* for keeping the number of keys 1, and Processors 2, 3 and 4 use *H(2)*, *H(3)* and *H(4)* respectively. That means a key 1 on a smaller-numbered processor has a smaller rank. In Step (2), H is redistributed in the "block" format and every processor computes a local running sum (figure (b)), then offsets it by total sums of preceding processors (figure (c)). For example, Processor 3 computes its local running sum *{0, 3, 4}*, and offsets all the elements by total sums of Processor 1 and 2, that is, *9* (*=6+3*). The resulting sum will be *{9, 12, 13}*. In Step (3), H is redistributed back in the "cyclic" format and every processor computes ranks of their own keys (figure (d)).

5.2 Considerations

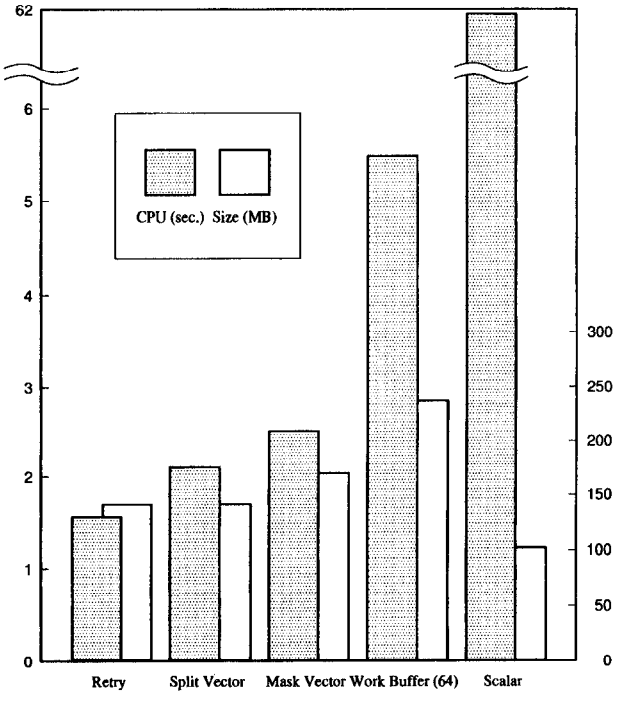
This method is fully parallelized and therefore efficient. The amount of computation per processor in Step (2) is constant even when the number of processors changes, therefore Step (2) could be a bottleneck of performance improvement in highly parallel machines.

6. EVALUATION RESULTS

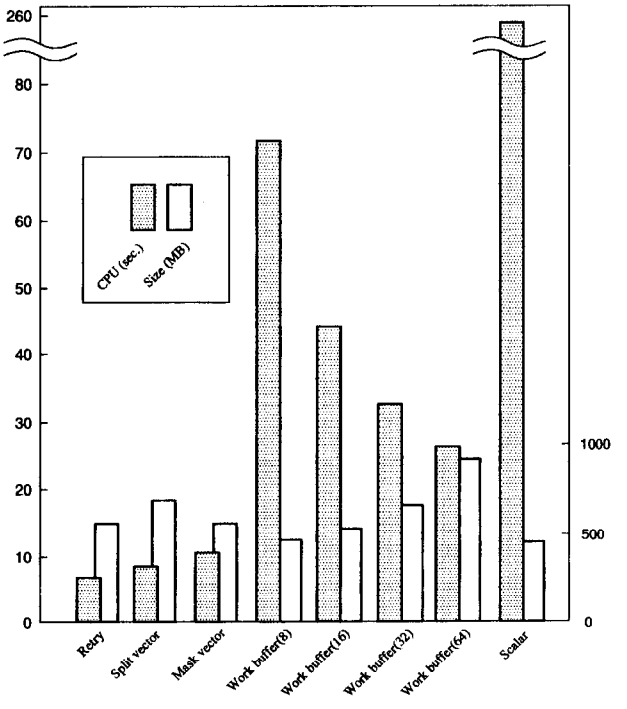
We implement the methods described above to the NPB 1.0 IS program on an NEC SX-4 shared-memory vector parallel supercomputer using the FORTRAN77/SX compiler. This section shows some results of the evaluation.

6.1 Comparison of Vectorization Methods

Figure 7 and Table 1 show the comparison of the retry method, the split vector method, the mask vector method, and the work buffer method[2] for an example of existing methods. The parenthesized numbers for the work buffer method denote vector lengths. The work buffer method with a long vector needs a



(a) Results of NPB IS Class A



(b) Results of NPB IS Class B

Figure 7: Comparison of Vectorization Methods

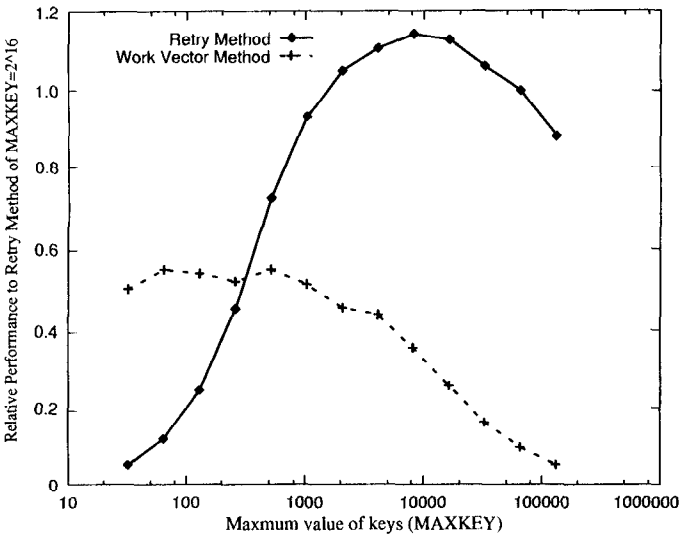


Figure 8: Evaluation of Conflict Frequency

huge amount of memory and the longest vector length we could implement was 64. We use a vector length of 256 for the other vector methods.

The methods described in this paper all show better performance than the work buffer methods. The retry method is the best of three new methods, and better than the 64-way work buffer method in class B.

6.2 Distribution of Input Keys

The performance of the retry method depends on the input data. We measured the execution time of the retry method version varying conflict frequency; we fix the number of keys to 2^{16} and vary the maximum value of keys. We also measured the execution time of the 64-way work buffer version under the same conditions.

Figure 8 shows the results. Although the performance of retry method gets worse when the conflict frequency becomes higher, it is better than work buffer method except at the very high frequency area.

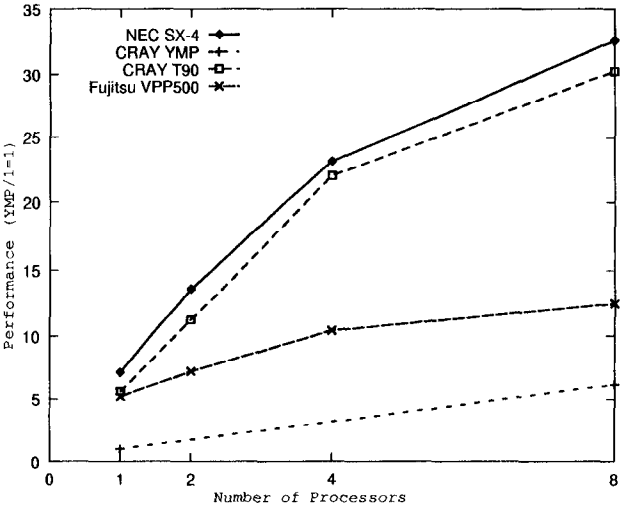
6.3 Evaluation of the Parallelization Method

Figure 9 and Table 2 show the results of the parallelized retry method version. The table also shows the official results of CRAY Y-MP, C90, T916 and Fujitsu VPP700[4]. With the methods described in this paper, the SX-4 got the world's fastest results (in November 1996) of class B.

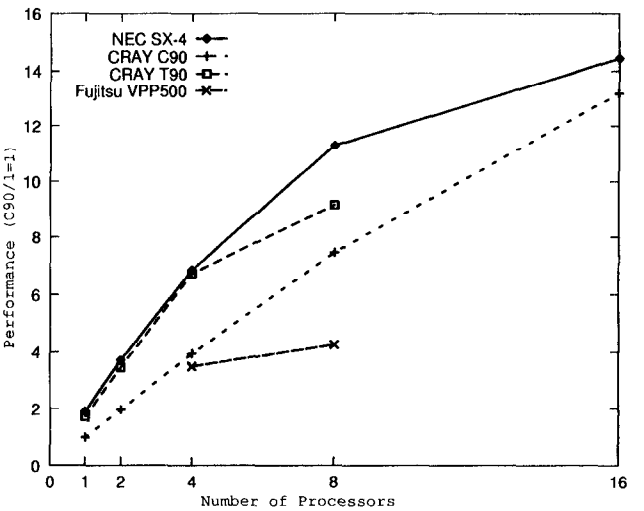
7. CONCLUDING REMARKS

We developed new fast integer sorting methods for vector computers and shared-memory parallel vector computers. Using these methods, we got the world's fastest results of the NPB IS program on the NEC SX-4. Our methods are also applicable to a wide range of applications such as "particle pusher" codes.

We now plan to more tune up our sorting programs to get better performance, and try more input patterns or more different



(a) Results of NPB IS Class A



(a) Results of NPB IS Class A

Figure 9: Evaluation of NPB IS Class B

methods. We also plan to apply our methods to real applications such as particle simulations.

ACKNOWLEDGMENTS

We would like to thank Toshiyuki Nakata for his valuable remarks and discussions about sorting algorithms.

We also thank Takeo Fujimori for his help in the experiments on the SX-4.

REFERENCES

[1] Bailey, D. et al. The NAS Parallel Benchmarks, RNR Technical Report RNR-94-007, Mar. 1994.

[2] Elton, B. H. and Miura, K. A Vector-Parallel Implementation and Statistical Analysis of the Bucket Sort on a

Vector-Parallel Distributed Memory System: Lessons Learned in the Integer Sort NAS Parallel Benchmark, Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, pp.782–3, 1995.

[3] Ishiura, N. et al. Sorting on a vector processor (in Japanese), Transactions of the Information Processing Society of Japan, vol.29, no.4, pp.378–85, 1988.

[4] Saini, S. and Bailey, D. H. NAS Parallel Benchmark (Version 1.0) Results 11-96, Report NAS-96-18, NASA Ames Research Center, Nov. 1996.

Method	CPU (sec.)	Ratio	Size (MB)
Retry	1.55	39.96	140
Split vector	2.09	29.63	140
Mask vector	2.48	24.96	172
Work buffer (64)	5.50	11.26	236
Scalar	61.94	1.00	104

(a) Results of NPB IS Class A

Method	CPU (sec.)	Ratio	Size (MB)
Retry	6.77	38.00	560
Split vector	8.49	30.30	688
Mask vector	10.60	24.27	560
Work buffer (8)	71.55	3.60	468
Work buffer (16)	44.11	5.83	532
Work buffer (32)	32.48	7.92	660
Work buffer (64)	26.20	9.81	916
Scalar	257.27	1.00	456

(b) Results of NPB IS Class B

Table 1: Comparison of Vectorization Methods

Number of CPUs	1	2	4	8
NEC SX-4 (sec.)	1.55	0.81	0.45	0.29
(Ratio to Y-MP/1)	7.39	14.14	25.46	39.51
Cray Y-MP	11.46	----	----	1.85
	1.00	----	----	6.19
Cray T916	2.02	1.02	0.52	0.38
	5.67	11.24	22.04	30.16
Fujitsu VPP700	2.3968	1.8038	1.2519	1.1249
	4.78	6.35	9.15	10.19

(a) Results of NPB IS Class A

Number of CPUs	1	2	4	8	16
NEC SX-4 (sec.)	6.77	3.50	1.877	1.139	0.896
(Ratio to Y-MP/1)	1.91	3.69	6.88	11.34	14.42
Cray C90	12.92	6.50	3.30	1.73	0.98
	1.00	1.99	3.92	7.47	13.18
Cray T916	7.44	3.74	1.92	1.41	----
	1.74	3.45	6.73	9.16	----
Fujitsu VPP700	9.1964	6.0875	4.1363	3.1959	2.7231
	1.41	2.12	3.12	4.04	4.74

(b) Results of NPB IS Class B

Table 2: Evaluation of NPB IS Class B