A method of vector processing for shared symbolic data *

Yasusi Kanada

Real-World Computing Partnership, Takezono 1-6-1, Tsukuba, Ibaraki 305, Japan.
kanada@trc.rwcp.or.jp
Fax: +81-298-53-1642

Abstract

Kanada, Y., A method of vector processing for shared symbolic data, Parallel Computing, X (1993) XXX-XXX.

Conventional processing techniques for pipelined vector processors such as the Cray-XMP, or data-parallel computers, such as the Connection Machines, are generally applied only to independent multiple data processing. This paper describes a vector processing method for multiple processings including parallel rewriting of dynamic data structures with shared elements, and for multiple processings that may rewrite the same data item multiple times. This method enables vector processing when entering multiple data items into a hash table, address calculation sorting, and many other algorithms that handle lists, trees, graphs and other types of symbolic data structures. This method is applied to several algorithms; consequently, the performance is improved by a factor of ten on a Hitachi S-810.

Keywords. symbol processing; vector processing; hashing; sorting.

1. Introduction

An attached vector processor, the Hitachi M-680H IDP (Integrated Database Processor) [10], is designed for database processing and has been applied to several symbolic processing applications [14]. However, most other vector processors, such as the Cray-XMP or Hitachi S-3800, are mostly used for numerical processing, and are rarely used for symbolic processing. One of the reasons that the extension of applications from numerical to non-numerical areas has been prevented is that no vectorization method has yet been established that is widely applicable to processing dynamic data structures connected by pointers, such as linear lists, trees and graphs.

The symbolic vector-processing methods developed by Kanada, et. al. [5, 6, 7] enable vector processing of multiple dynamic data structures by *vectorization*, a program transformation. When these methods are applied, the data structures are accessed through *index vectors,* which contain pointers or indices to the data to be processed. These methods are called *simple index-vector-based vector-processing methods* (SIVP) in this paper. The list-vector-processing facility and conditional control facilities [4], such as masked operations, of vector processors are used in SIVP.

However, conventional vector-processing methods including SIVP are basically applied only to *independent* multiple data processings. This means that these methods cannot vectorize multiple processings including rewriting of data with shared elements, such as graphs, and they cannot vectorize multiple processings that may rewrite the same data item multiple times, such as entering data items into a hash table (See Section 2). The *filtering-overwritten-label method* (FOL) explained in this paper solves this problem.

The above problem is explained further in Section 2. The principle and algorithms of FOL are shown in Section 3. Several applications and performance evaluations of FOL are shown in Section 4. Related works are mentioned briefly in Section 5.

2. Problems in Vector Processing of Shared Data

In the symbolic vector-processing methods shown in Kanada [5, 6, 7], data items are read and written through index vectors. **Figure 1** illustrates two types of index vectors: a vector of pointers to the data items, and a vector of subscripts or displacements of the data items. Using index vectors, parts of the symbolic data are gathered into a vector register or scattered to main storage by the so-called list-vector instructions or indirect vector load/store instructions.

The classes of vectorizable and unvectorizable processings are explained in **Figure 2**. SIVP is basically applicable only for independent multiple data processings. This means that SIVP can be applied to an index vector that contains pointers or indices to independent data items (Figure 2a). However, SIVP can also be applied to *read-only* processings of multiple data items including shared ones. The index vector may have pointers to the same data items (Figure 2b), as long as it does not update the data. But SIVP cannot be applied for *rewriting* multiple data items with sharing (Figure 2b), because if it were applied, the processings that must be performed sequentially would be performed in parallel. This may give incorrect results because the processing order of the elements is not defined in vector processors with parallel pipelines, such as S-3800. Therefore, SIVP cannot be applied for rewriting partially shared data structures (illustrated in **Figure 3**), because read-only vector-processable index vectors (Figure 2b) may be generated while processing such data structures.Two examples that cannot be processed by SIVP methods are shown below (in **Figure 4**). The first example shows multiple data items being entered into a hash table. This processing is called *multiple hashing* in this paper. The entered data items are chained from the hash table entries. Figure 4a shows sequential processing in which two keys are being entered. The numbers above the arrows indicate the order of execution. Keys 353 and 911 are entered in this order. Because the hashed values of these keys are both five, they collide. So they are chained from the same hash table entry. Figure 4b shows the problem of multiple hashing by *forced* vector processing. The keys are initially stored in a vector, and the hashed values are calculated by data-parallel operations, or vector operations, and are stored into another vector that is used as an index vector in the following process. If no collisions occur, the key writing is processed properly. However, collisions make correct processing impossible. The pointer to the second key overwrites that to the first key in this figure.

The second example is tree rewriting, which transforms an input tree into an equivalent final form by applying a rewriting rule. **Figure 5** illustrates two ways of rewriting an operation tree; here, the associative law is used as the rewriting rule. The associative law is expressed as $X * (Y * Z) \rightarrow (X * Y) * Z$. The arrow indicates the direction of rewriting. The input tree is $a * (b * (c * d))$ in Figure 5. The rewriting is applied to nodes $n3$ and $n5$ in Figure 5a, and to nodes $n1$ and $n3$ in Figure 5b. Node $n3$ is “shared” between these two rewritings. A *forced* parallel rewriting by vector processing causes a tree that is not equivalent to the original to be generated, or the rewriting process to be aborted because of a nonexistent (phantom) node access. In this example, multiple (two) nodes are rewritten in a unit process, i.e., one rewriting.

3. A Solution: Filtering-overwritten-label Method

A method of rewriting multiple symbolic data items with sharing, called the *filtering-overwritten-label method* (FOL), has been developed. The principle and algorithms of FOL are explained in this section.

3.1 The principle of FOL

The problem explained in Section 2 is solved by the decomposition of a data set into parallel-processable subsets, as illustrated in **Figure 6**. In this principal method, the element set (S) of the original index vector is split into parallel-processable sets ($S1$, $S2$, and $S3$) and is restored into parallel-processable index vectors ($V1$, $V2$, and $V3$). These index vectors are processed separately by vector operations. Thus, the elements of a vector are processed in parallel and the vectors are processed one by one.[1] Pointers to one data item, a for example, are scattered into different index vectors. This principal method processes the *unshared* part of the multiple data items in parallel and the shared part sequentially.

To implement the above principal method, a method of decomposing data sets into parallel-processable sets must be developed. The data that cannot be processed in parallel are multiple pointers or indices that point to *one* data item, and they can be detected by comparing all of the pairs of pointers or indices. However, this process needs $O(N^2)$ comparisons, so it will decrease performance. FOL[2] is a method of decomposing a data set in $O(N)$ time in normal cases by vector processing.

Before explaining the algorithms of FOL in the following subsections, a method of multiple hashing using FOL is explained informally (See **Figure 7**)[3]. Only the keys are entered into the hash table in this example for the sake of simplicity. The keys to be entered are initially stored in a vector. Each hash table entry has a work area for storing *labels*.

The detection of collisions (See FOL processes 1 and 2 in Figure 7) is performed entirely by vector operations in the following manner. In Step 1, the subscripts of the key vector are written into the work area indexed by the hashed values. These subscripts are called *labels* in FOL. In Step 2, the labels are read immediately after writing, using the same indices, i.e., the hashed values. These label writing and reading processes are performed using the list-vector instructions. The elements of the read vector are compared with the original labels. They are equal if there are no collisions. However, they are not equal when there are collisions, because collisions cause overwriting of labels in the work areas. The results of the comparisons are stored into a mask vector, a Boolean vector. The key vector elements, whose corresponding mask vector elements are *true*, are parallel-processable data items. In the example in Figure 7, the second to fourth elements of the key vector form the first parallel-processable set of keys, because the second to fourth elements of the mask vector are *true*. These keys are entered into the hash table in parallel in Step 3. In Step 4, the above process is repeated until all of the keys are classified into a parallel-processable set, and all of them are successfully entered into the hash table.

3.2 FOL for rewriting single data item per unit process

FOL is a generalization of the multiple hashing method explained in the previous subsection. FOL can be applied to a wide range of processings that rewrite multiple data items with possible sharing. The FOL algorithm for rewriting a single data item per unit process to be vectorized is shown in this subsection. An extension of FOL for rewriting multiple data items per unit process, such as rewriting the operation tree shown in Section 2, is shown in the next subsection.

An FOL algorithm that decomposes a set of data items into parallel-processable sets is shown below. The whole process of this algorithm can be performed by vector operations on a vector processor such as the Hitachi S-3800.

n **Algorithm: Filtering-overwritten-label Method 1 (FOL1)**

o **Input**

This algorithm inputs an index vector V. The elements of V are pointers or indices to storage areas containing data items $d_1, d_2, \ldots, d_N$, where there may be duplicated data items (i.e., the same data items may appear multiple times in the sequence. The storage area pointed to by v, an element of V, is denoted by $v\rightarrow$, and the data item stored in $v\rightarrow$ is denoted by $v\rightarrow d$.

o **Output**

This algorithm outputs the sets of parallel-processable data items $S_1, S_2, \ldots, S_M$ (the value of M is obtained as a result), where $\bigcup_{i=1}^{M} S_i = \{d_1, d_2, \ldots, d_N\}$ and $S_1, S_2, \ldots, S_M$ are disjoint sets, i.e., $S_i \cap S_j = \emptyset$ for arbitrary i and j (***disjoint decomposition condition***).

o **Processing conditions**

- Processing P_i ($i = 1, 2, \ldots, N$), which corresponds to the main processing in Figure 7, is applied to data item d_i by vector operation after the execution of this algorithm. All the data items included in an output set S_j ($j = 1, 2, \ldots, M$) may be processed in parallel or in an arbitrary order in processing P_i.[4] The order of processing for each output set can be arbitrary, but two data items belonging to different output sets may not be executed in parallel. So, any two output sets must be processed sequentially. The execution order between the processings of two arbitrary data items must not affect the correctness of the result.[5]
- If va and vb are arbitrary elements of V before execution, $va\rightarrow$ and $vb\rightarrow$ may possibly be in the same storage area. However, the data items pointed to by va and that pointed to by vb are regarded as non-identical data items in the algorithm description.

- For each element v of the index vector V, a work area denoted by $v{\to}w$ is allocated in storage area $v{\to}$. This means that, if $va{\to}$ and $vb{\to}$ are the same storage area, $va{\to}w$ and $vb{\to}w$ are also the same work area even when va and vb are not identical.

o **Procedure**

(0) Preprocessing: Set 1 to variable j. Assign a unique label to each element of V. The labels may be assigned before the execution time if possible.[6]

(1) Writing labels: Write the labels of $v_1, v_2, \ldots, v_n$ into the work areas $v_1{\to}w$, $v_2{\to}w$, …, $v_n{\to}w$; where n is the number of elements of V. The execution order is arbitrary, and the labels may be processed in parallel.

(2) Detection of overwriting: Read the labels from work areas $v_1{\to}w$, $v_2{\to}w$, …, $v_n{\to}w$, and compare them with the labels of $v_1, v_2, \ldots, v_n$, respectively. Step 1 must be completed before reading the labels. (Synchronization may be necessary.) If the value read from $v_i{\to}w$ is not equal to the label of v_i, v_i has been overwritten. Then, the set of all data items pointed to by the elements of V except the data items that failed the equality check forms the parallel-processable set. This set is named S_j. More exactly, set $\{u_1{\to}d, u_2{\to}d, \ldots, u_m{\to}d\}$ is assigned to S_j, where u_k ($k = 1, 2, \ldots, m$) are all the elements of V, which satisfies the relation $u_k{\to}w = l_k$; where l_k is the label for u_k.

(3) Updating control variables: Add 1 to j. Delete the pointers or indices pointing to data items in S_j, from V. The number of elements in V is reduced, thus the value of n is changed.

(4) Repetition: Repeat Steps 1 to 3 above until V becomes empty. When terminated, set j – 1 to variable M. n

In the example of multiple hashing shown in Subsection 3.1, the main processing (entering of keys) is amalgamated to the steps of FOL for efficiency, but the main processing is not included in FOL1 (the above algorithm), to make the algorithm multi-purpose.

The following condition must hold for the sake of the correctness of FOL1.

n **The exclusive label storing condition (ELS condition)**

One of the multiple labels written into one work area is stored correctly. This means that if two labels la and lb are written into the same area in parallel, the stored value is not an amalgam of la and lb. Which one of these labels is successfully stored is arbitrary. n

The ELS condition is guaranteed when the label length is equal to or less than the machine word length in normal pipelined vector processors. This condition holds hereafter.[7]

The lemmas and theorems in FOL1 are shown below.

n **Theorem 1: Termination property**

Algorithm FOL1 terminates.

o **Proof**: Any label read from a work area is equal to one of the original labels written to the work area by the ELS condition. Thus, the read label is equal to at least one of the original labels in Step 2, and S_j is not an empty set for arbitrary j. This means that the number of elements in the index vector V is reduced every time in Step 3. Therefore, V always becomes empty in finite iterations, and FOL1 terminates. n

The correctness of FOL1 will be proved using two lemmas.

n **Lemma 1: Disjoint decomposition**

The disjoint decomposition condition shown in the output specification of FOL1 holds. That means, the union of the output sets, S_1, S_2, …, S_M, is equal to the input data set and $S_i \cap S_j = \varnothing$ for arbitrary i and j when FOL1 terminates.

o **Proof**: If the following conditions hold, the disjoint decomposition condition holds:
(a) each element of the output set S_j (j = 1, 2, …, M) is equal to an input data item,
(b) each input data item is equal to an element of an output set, and
(c) the output sets are disjoint.

First, we show condition (a). The index vector V consists of all the pointers, each of which points to an input data item at the beginning of execution. An element, e, of an arbitrary output set S_j is is equal to an input data item, because it is selected from all of the data items that the elements of V point to in Step 2.

Next, we will show conditions (b) and (c). V consists of all the pointers or indices to the input data items at the beginning of execution but to no others. V becomes an empty set at termination. The elements are deleted from V in Step 3 immediately after they are added to one of the output sets S_j (j = 1, 2, …, M) in Step 2, and all the elements are deleted from V before execution terminates. That means that all of the data items pointed to from V are added to one of the output sets. In addition, the pointer or index to an element of S_j is deleted immediately after the content of S_j is computed, so this pointer or index never belongs to S_m ($m > j$). Thus, the output sets are disjoint. n

The cardinalities (the numbers of elements) of S_1, S_2, …, S_M are denoted by $|S_1|$, $|S_2|$, …, $|S_M|$. Then, the following lemma and theorem hold.

n **Lemma 2**

If d_k and d_l ($k \neq l$) are arbitrary elements of an output set S_k, then d_k and d_l are in different areas (d_k and d_l are different data items).

o **Proof**: The lemma is proved by contradiction. If d_k and d_l are in the same area, the pointers or indices to them, which are the elements of V, have the same value. The labels assigned to these elements are stored into the same work area. One of these labels is overwritten by the other, so d_k and d_l are included in different output sets in Step 2. This is a contradiction, so the lemma is concluded. n

n **Theorem 2: Correctness**

The output conditions hold when FOL1 terminates.

o **Proof**: This theorem is proved by Lemmas 1 and 2. (Lemma 2 guarantees that the output sets are parallel-processable.) n

One more theorem is given but the proof is omitted.

n Theorem 3

The following relation always holds: $|S_1| \geq |S_2| \geq \ldots \geq |S_M|$, and $M = 1$ (the number of iterations in FOL1 is equal to one) when the input data does not have duplicates. n

The allocation of the work area used in FOL1 is explained below. Normally, the work area can share storage with the area used for main processing. This is because it does not matter whether the value held in the area pointed to by the elements of V is destroyed before FOL1 is applied by writing labels, and because there is no possibility that the wrong value, which is not a correct label, is read in the process of overwriting detection. It does not matter whether the value is destroyed because main processing will rewrite the storage area where the labels are written by FOL1. Conversely, the condition that the main processing always rewrites the work area, where the labels are written, or a weaker condition must hold. There is no possibility that the wrong value is read, because there is no possibility that, while reading labels in Step 2, the labels are read from an area where no labels were written, because the ELS condition holds and the labels are read through the same pointers or indices used when writing the labels.

Because the size of each work area is $\log_2 N$ bits or more, the shared area must be extended when the main processing requires less area. The size must be $\log_2 N$ bits or more because the work area must have enough capacity to hold one of N different labels.

The performance of FOL1 is examined next. The sequentially processed part of main processing is not accelerated by FOL. On the contrary, the execution of this part becomes slower because of the overhead of parallel-processable data detection. Consequently, the sequential execution is better than FOL in processing where most of the data items cannot be processed in parallel. However, if sharing rarely occurs and most of the data items can be processed in parallel, FOL is promising.

The following theorem guarantees that the execution time of FOL1 is $O(N)$ when the amount of sharing is small.

n Theorem 4: *O*(*N*) execution time

If condition $|S_1|$ » $\sum_{i=2}^{M} |S_i|$ holds, then the execution time of FOL1 is $O(N)$.

o **Proof**: The execution time of Steps 1, 2 and 3 in the j-th iteration is approximately in proportion to $\sum_{i=j}^{M} |S_i|$, the number of elements of V ($j = 1, 2, \ldots, M$). Thus, if the above condition holds, the execution time of the second and later iterations can be ignored compared with that of the first iteration. Then the execution time of FOL1 is $O(N)$, because the execution time of the first iteration is approximately in proportion to N, the number of elements of V in the initial state, because $\sum_{i=1}^{M} |S_i| = N$. n

In particular, the execution time of FOL1 is $O(N)$ when there are no duplicates in the input data, by Theorems 3 and 4.

A lemma and two theorems are given. Theorem 5 guarantees the best performance in a sense, when condition $|S_1|$ » $\sum_{i=2}^{M} |S_i|$ does not hold.

n **Lemma 3**

If there are *M’* duplicates in the input data, all of which are the same, including the original data (i.e., there is a storage area that is shared by *M’* input data), and if there are no more than *M’* duplicates, the number of output sets, *M*, is equal to *M’*.

o **Proof**: If there are *M’* duplicates, *V* has *M’* elements, which point to the same storage area containing the duplicated data, at the beginning of FOL1. As explained in the proof of Theorem 1, there always exists a label that coincides with the original in Step 2. In addition, each input data item has a unique label and only one of the labels is read repeatedly *M’* times, so, in the labels of the *M’* elements of *V*, there is exactly one label that coincides. Therefore, exactly one of these elements is deleted from *V* every time Step 3 is executed. The number of iterations in FOL1 is equal to *M*. Thus, *M’* is equal to *M*. n

n **Theorem 5: Minimum decomposition**

If $T_1 \cup T_2 \cup \ldots \cup T_{M''}$ is an arbitrary decomposition of the input data, i.e., $\bigcup_{i=1}^{M''} T_i = \{d_1, d_2, \ldots, d_N\}$, where the element of T_i is parallel-processable for arbitrary *i*, and the number of output sets of FOL1 is *M*, then $M'' \geq M$. That means that the number of output sets of FOL1 is the minimum.

o **Proof**: The duplicated data item does not belong to the same output set because, if it did, the output set would not be parallel-processable. Thus, if there are *M’* duplicates in the input data, the number of parallel-processable sets is no less than *M’* for arbitrary decomposition. The number of output sets of FOL1 is also *M’* by Lemma 3. Thus, the number of output sets of FOL1 is the minimum. n

n **Theorem 6: Worst execution time**

The execution time of FOL1 is $O(N^2)$ when the following condition holds: $|S_1| = |S_2| = \ldots = |S_M| = 1$.

o **Proof**: The above condition means that the number of elements of *V* decreases one by one. So the total execution time of the *j*-th iteration is $(N - j + 1)\, t + c$, if *c* is a constant and the sum of the execution time for a vector element in an iteration is *t*, because the total execution time is approximately in proportion to the number of vector elements, and the total execution time is $\sum_{j=1}^{N} (N - j + 1)\, t + C$, which means it is $O(N^2)$. n

The application area of FOL1 is considered next. FOL1 is a vectorization/parallelization method that can be used in a wide range of applications. Multiple hashing is a typical multiple data processing where a small number of shared items exist, but the same condition holds in many applications, for example, address calculation sorting and parallel rewriting of lists, trees (DAGs) or graphs with sharing, as illustrated in Figure 3.

Finally, a method of improving the execution time of FOL1 by simplifying its process is examined. The following simplified method can be applied when there is no duplication in the values to be written into the area pointed to by the elements of the index vector V. In this case, these values can be used for labels, and the label writing and the main processing (i.e., writing the values) can be performed at the same time. For multiple hashing, the above condition holds when the keys are unique and are used as labels.

3.3 FOL for rewriting multiple data items per unit process

FOL1 can be applied when only one data item is rewritten in a unit process. An FOL algorithm, which is applied when multiple data items are rewritten in a unit process, is as follows.

n **Algorithm: Filtering-overwritten-label Method 2 (FOL*)**

o **Input**

This algorithm inputs index vectors $V_1, V_2, \ldots, V_L$, which have the same number of elements. The elements of these vectors are pointers or indices to storage areas containing data items $d_{i1}, d_{i2}, \ldots, d_{i\,L}$ ($i = 1, 2, \ldots, N$), where there may be duplicated data items. The storage area pointed to by v, an element of V_k, is denoted by $v\!\to$, and the data item stored in $v\!\to$ is denoted by $v\!\to\!d$.

o **Output**

This algorithm outputs sets of tuples $\langle d_{i1}, d_{i2}, \ldots, d_{iL}\rangle$($i = 1, 2, \ldots, N$) of parallel-processable data items: $S_1, S_2, \ldots, S_M$ (the value of M is obtained as an execution result of this algorithm), where $\bigcup_{j=1}^{M} S_j = \{\langle d_{i1}, d_{i2}, \ldots, d_{iL}\rangle \mid i = 1, 2, \ldots, N\}$ and $S_1, S_2, \ldots, S_M$ are disjoint sets, i.e., $S_i \cap S_j = \varnothing$ (***disjoint decomposition condition***).

o **Processing conditions**

- Processing P_i ($i = 1, 2, \ldots, N$) is applied to a data tuple $\langle d_{i1}, d_{i2}, \ldots, d_{iL}\rangle$ by vector operation after the application of this algorithm. All the data items included in an output set S_j ($j = 1, 2, \ldots, M$) may be processed in parallel or in any order, but any two data items belonging to different output sets may not be executed in parallel. The execution order must not affect the correctness of the result.
- If va is an arbitrary element of V_f and vb is an arbitrary element of V_g before the execution where $f, g \in \{1, 2, \ldots, L\}$, $va\!\to$ and $vb\!\to$ are possibly the same storage area. This means that $V_1, V_2, \ldots, V_L$ may have pointers with the same values as their elements.
- Work areas to be used in this algorithm are reserved for each storage area pointed to by each element of $V_1, V_2, \ldots, V_L$. The work area pointed to by v is denoted by $v\!\to\!w$.

o **Procedure**

(0) Preprocessing: Set 1 to variable j. Assign a unique label to each element of index vectors V_k ($k = 1, 2, \ldots, L$). That means, if la is the label of an arbitrary element of V_{k1} and lb is the label of an arbitrary element of V_{k2} and these elements are different, condition $la \neq lb$ must hold. The labels may be assigned before execution.

(1) Writing labels: $v_{k1}, v_{k2}, \ldots, v_{kn}$ are assumed to be the elements of the index vector V_k, where n is the number of elements of $V_1, V_2, \ldots, V_L$ (where all the vectors have the same number of elements). For $k = 1, 2, \ldots, L$, perform the following process. Write the labels of $v_{k1}, v_{k2}, \ldots, v_{kn}$ into the work areas $v_{k1}\rightarrow w$, $v_{k2}\rightarrow w$, …, $v_{kn}\rightarrow w$, respectively. The execution order is arbitrary, and the labels may be processed in parallel.[8]

(2) Detection of overwriting: For $k = 1, 2, \ldots, L$, perform the following process. Read the labels from work areas $v_{k1}\rightarrow w$, $v_{k2}\rightarrow w$, …, $v_{kn}\rightarrow w$ and compare them with the labels of $v_{k1}, v_{k2}, \ldots, v_{kn}$, respectively. Step 1 must be completed before label reading. If $v_i\rightarrow w$ is not equal to the label of v_i, it means that v_i is overwritten. The set of all data tuples $\langle d_{i1}, d_{i2}, \ldots, d_{iL}\rangle$ whose elements are the data items pointed to by the i-th elements of $V_1, V_2, \ldots, V_L$, respectively, except the tuples which contain data items, for which the equality check has failed, is the parallel-processable set S_j. More exactly, set S_j is defined as follows: $l_{1j}, l_{2j}, \ldots, l_{Lj}$ are assumed to be the labels assigned to the index vector elements $v_{1i}, v_{2i}, \ldots, v_{Li}$, then S_j is the set of all tuples $\langle v_{1i}\rightarrow d, v_{2i}\rightarrow d, \ldots, v_{Li}\rightarrow d\rangle$ $(i = i_1, i_2, \ldots, i_n)$, where condition $v_{1i}\rightarrow w = l_{1i} \wedge v_{2i}\rightarrow w = l_{2i} \wedge \ldots \wedge v_{Li}\rightarrow w = l_{Li}$ holds.

(3) Updating control variables: Add 1 to j. For $k = 1, 2, \ldots, L$, delete the pointers or indices pointing to data items in a tuple in S_j, from V_k. The number of elements in V_k is reduced for each k, thus the value of n is changed.

(4) Repetition: Repeat Steps 1 to 3 above until V_k becomes empty. (Testing only one of $V_1, V_2, \ldots, V_L$ is enough because they all have the same number of elements.) When terminated, set $j - 1$ to variable M. n

The ELS condition must hold in Step 1 as in FOL1. In addition, a deadlock may occur unless another appropriate condition is added to the label writing order, as explained in Step 1. This means that the relation in Step 2 does not hold for any element of the index vectors, if the writing order of a vector is not the same as others. Then, S_j may be an empty set and the control does not exit from the loop in FOL*.

A method to avoid the above problem is explained. The elements of the index vectors except the last ones are written by vector instructions in parallel, but the last elements are written by scalar instructions sequentially after the execution of the vector instructions. It is asserted that there are no shared elements among the last elements of the index vectors. Then, the relation shown in Step 2 holds at least for the last elements, and S_j is prevented from being empty. Thus, if the possibility that the relation does not hold for the elements processed by vector instructions is small enough, this method will be sufficient. However, this method may cause a significant decrease in parallelism when the possibility is not small, and the acceleration ratio may become less than 1. So, a better method should be developed.

The proofs of the termination property and the disjoint decomposition condition are omitted because they can be proved in the same way as the case of rewriting single data item shown in Subsection 3.2.

The performance of FOL* is examined next. When the number of data items rewritten in a unit process, i.e., L, is large, the execution time of FOL* becomes larger compared with the main processing, and the acceleration ratio[9] of the total process is low. Thus, FOL* is considered to be practical only when L is less than five or so. The value of L is two in the case of the operation tree rewriting shown in Section 2, so the vectorized algorithm will be practical in this case.

4. Applications of FOL

Three applications of FOL1 and the resulting performances are shown in this section.

4.1 Multiple hashing

There are two collision resolution methods in hashing: *open addressing* and *chaining* [11]. The algorithms previously shown in this paper apply chaining. The algorithm for multiple hashing using open addressing, which is based on a specialized version of FOL, is shown in **Figure 8**. This is an optimized version of a multiple hashing algorithm called the "overwrite-and-check method" [8]. The keys are used as labels in this algorithm, and, thus, only keys are contained in the hash table. Because all the labels must be different, all the keys must be different in this algorithm. Each unused entry in the hash table is initialized to a special value, *unentered*, which is not used as a key value. *unentered* is used to display whether the entry is used or not.

The algorithm is written in a language with a parallel array assignment statement and a **where** statement, such as Fortran 90. Each assignment in each parallel array assignment statement may be performed in parallel. However, no two statements can be executed in parallel, if the parallel execution may cause a wrong result. For example, if A = (1, 2, 3), B = (10, 11, 12), and M is a *mask vector* (a boolean vector used for controlling vector operations), and the value is (*true*, *false*, *true*), the following statement updates the value of A as (10, 2, 12);

```
    where M do A := B; end where;
```

This language also has a *countTrue* function and a **where** operator. If M is a mask vector, expression *countTrue*(M) returns the number of occurrences of *true* in M. For example, if M is array (*true*, *false*, *true*), then *countTrue* returns 2. Expression A **where** M means a vector of elements of A which correspond to *true* elements of M. For example, if A = (1, 2, 3) and M = (*true*, *false*, *true*), A **where** M returns (1, 3). Expression $A[x : y]$ in Figure 2 means a slice (subarray) of A, ($A[x]$, $A[x+1]$, …, A[y]).

The difference between the old and new algorithms lies in the subscript recalculation methods in the case of collisions. In the original algorithm, the n-element vector of subscripts, *hashedValue*[1 : n], is computed from the old subscripts, which are initially equal to the hashed values, as follows:

```
    hashedValue[1 : n] :=
        (hashedValue[1 : n] + 1) mod size(table);
            — All of the elements are incremented by one.
```

However, if three or more keys in the key vector collide, the keys that were not entered successfully cause collisions when tried to be reentered, because the subscripts are still the same. Thus the optimized algorithm recalculates the vector of subscripts as follows:

```
    hashedValue[1 : n] :=
        (hashedValue[1 : n] + (key[1 : n] & 31) + 1) mod size(table);
            — “&” means bitwise “and” operation.  It is asserted that size(table) > 32.
```

The optimized algorithm is coded in Fortran[10] and executed on Hitachi S-810 [13], which is an older-generation vector processor than the S-3800. All of the innermost loops are vectorized. **Figures 9** and **10** display the CPU time and acceleration ratio of multiple hashing when the table sizes are 521 and 4099. The horizontal axes show the load factor (the ratio of the filled table entries) after entering the keys. The acceleration ratio reaches the maximum value, 5.2 or 12.3, when the load factor is 0.5. The reason why the acceleration ratio increases when the load factor is less than 0.5 is that the vector length is proportional to the load factor. The reason why the acceleration ratio decreases when the load factor is between 0.5 and 1.0 is that the effect of reducing the performance caused by the increase in sequentiality is larger than the acceleration effect caused by the increase in vector length.

The results of the optimized algorithm shown in Figures 9 and 10 are better than those of the original [8] in that the acceleration ratio is larger when the load factor is between 0.5 to 0.98. This is the result of improving the method of subscript recalculation for colliding keys.

4.2 Address calculation sorting and distribution counting sort

There is a variation in address calculation sorting called the *linear probing sort* [3]. This algorithm uses a work array, *C*. Data are “hashed” and stored into *C*. The “hashing function” has the following property.

$$data[i] \le data[j] \Rightarrow hash(data[i]) \le hash(data[j]) \qquad (1 \le i \le n,\ 1 \le j \le n)$$

Because of this property, it is not really a hashing function, but an FOL technique that can be applied in the same way as multiple hashing. The order of data items stored in array *C* is sorted because of this function, if it is not disordered by the processing of colliding data items. The data in *C* are not contiguously stored, so they are packed into another array. The original array, *data*, can be used for this purpose. The algorithm is shown in **Figure 11**.

This algorithm can be vectorized using FOL1. The vectorized algorithm is shown in **Figure 12**. The data items to be sorted are asserted as non-negative here. This program consists of six parts, A through F. They correspond to the same name parts in the scalar algorithm except E, which is specific to vector processing based on the FOL method. The new data items are inserted into the sorted array *C* in part C. However, if old data items are already stored in these places, they are saved to array *work* in part C and restored to the next available places of *C* in part D.

There are two major differences between the hashing used in this address calculation sorting and the multiple hashing shown in the previous subsection. One difference is that the unique identifiers are used as labels instead of keys in the case of the multiple hashing. The assertion that the data items are non-negative is necessary because of the sharing of arrays between identifiers and the data items to be sorted, but this assertion can be eliminated if a different array is used for each purpose.

The other difference is the processing of colliding data items. Colliding data items must be inserted in an appropriate place in the sequence of sorted data. An attempt is made to insert all the colliding data items in parallel using vector operations. All unused entries in *C* are initialized to a special value, *unentered*, which is greater than any data value. This makes the above insertion possible.

Figure 13 shows an example of the address calculation sorting process compared to the original sequential sorting process. Though this algorithm uses a lot of local arrays, the size of these arrays, except *C*, can be remarkably reduced by an optimizing transformation. However, *C* must be at least twice as large as *n*.

The distribution counting sort [11] can also be vectorized using the *overwrite-and-check* technique. The vectorized distribution counting sort algorithm is omitted here.

A summary of the results is shown in **Table 1**. The maximum acceleration ratio of the address calculation sorting is more than ten.

4.3 Entering multiple data items into a binary tree

An algorithm that enters multiple data items into a *linked* binary tree using FOL1 has been developed. The tree is not balanced in this algorithm. **Figure 14** displays the result of a performance evaluation on the S-810. The horizontal axis indicates the number of uniformly random keys entered into the tree, and the vertical axis indicates the acceleration ratio. The tree has *Ni* elements, each of which contains a random key, before entering the keys. The reason why an empty tree is not used for benchmarking is that it is too disadvantageous for vector processing because all the keys to be entered create conflict when the tree is empty. The number of trials for each plotted point is only one, so this result is not very reliable. However, we can conclude that the average acceleration ratio is more than 1, though it is not a factor of ten, if the initial tree size is not very small and the amount of data to be entered is not very small.

5. Related Works

Appel and Bendiksen's vectorized garbage collection algorithm [1] implicitly includes a very specialized version of FOL. Suzuki et. al. [12] use a similar technique in the vectorized LSI routing algorithm. In both algorithms, the first output set S_1 is implicitly computed. The output sets S_2, S_3, …, S_M are not computed because they are unnecessary in these algorithms.

6. Conclusion

The *filtering-overwritten-label method* (FOL) given in this paper enables vector processing over a wide range of applications which include rewriting data with shared elements or possibly rewriting the same data item multiple times. The evaluation results show that the application of FOL to multiple hashing and to an address calculation sorting accelerates execution by a factor of ten. FOL is a promising vector-processing technique for processing lists, trees, graphs or other symbolic processings on pipelined vector processors or data-parallel computers such as the Connection Machines.

The main focus of future works will be to apply FOL to various symbolic algorithms including tree rebalancing and graph rewriting.

Acknowledgments

The author wishes to thank Dr. Sakae Takahashi of Hitachi Ltd. and Seiichi Yoshizumi, for their continuing support of his research, Shun'ichi Torii of Hitachi Ltd., for his comments that led to significant improvements in this paper, and Dr. Michiaki Yasumura of Keio University, Keiji Kojima, and Masahiro Sugaya of Hitachi Ltd. for their suggestions.

References

[1] Appel, A. W., and Bendiksen, A., Vectorized Garbage Collection, *J. Supercomputing* 3 (1989) 151–160.

[2] Flores, I., Computer Times for Address Calculation Sorting, *J. ACM*, Vol. 7, No. 4 (1960) 389–409.

[3] Gonnet, G. H., *Handbook of Algorithms and Data Structures*, Addison-Wesley (1984).

[4] Kamiya, S., Isobe, F., Takashima, H., and Takiuchi, M., Practical Vectorization Techniques for the "FACOM VP," *Information Processing '83* (1983) 389–394.

[5] Kanada, Y., Kojima, K., and Sugaya, M.: Vectorization Techniques for Prolog, *1988 ACM Int. Conf. on Supercomputing* (1988) 539–549.

[6] Kanada, Y., and Sugaya, M., Vectorization Techniques for Prolog without Explosion, *Int. Joint Conf. on Artificial Intelligence '89* (1989) 151–156.

[7] Kanada, Y., and Sugaya, M., A Vector Processing Method for List Based on a Program Transformation, and its Application to the Eight-Queens Problem, *Transactions of Information Processing* 30: 7 (1989) 856–868, in Japanese.

[8] Kanada, Y., A Vectorization Technique of Hashing and its Application to Several Sorting Algorithms, *PARBASE-90*, IEEE (1990) 147–151.

[9] Kanada, Y., A Method of Vector Processing for Shared Symbolic Data, *Supercomputing '91*, (1991) 722–731.

[10] Kojima, K., Torii, S., and Yoshizumi, S., IDP — A Main Storage Based Vector Database Processor, *1987 Int. Workshop on Database Machines* (1987) 60–73.

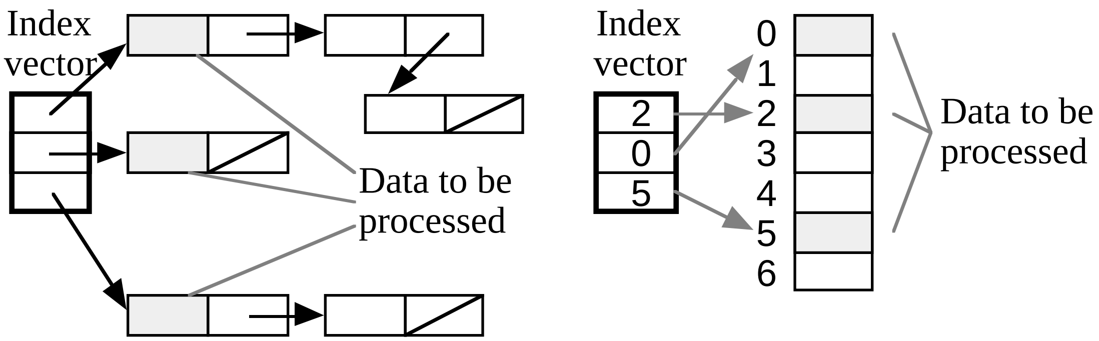
[11] Knuth, D. E., Sorting and Searching, *The Art of Computer Programming*, Vol. 3, Addison-Wesley (1973).

[12] Suzuki, K., Miki, Y., and Takamine, Y., *An Acceleration of Maze Algorithm Using Vector Processor*, Technical Report CAS 91-17 91:5, Institute of Electronics, Information and Communication Engineers (1991) 23–28, in Japanese.

[13] Nagashima, S., et al., Design Consideration for High-Speed Vector Processor: S-810, *IEEE Int. Conf. on Computer Design* (1984) 238–242.

[14] Torii, S., Kojima, K., Kanada Y., Sakata, A., Yoshizumi, S., Takahashi, M., Accelerating Non-Numerical Processing by An Extended Vector Processor, *4th Int. Conf. on Data Engineering* (1988) 194–201.

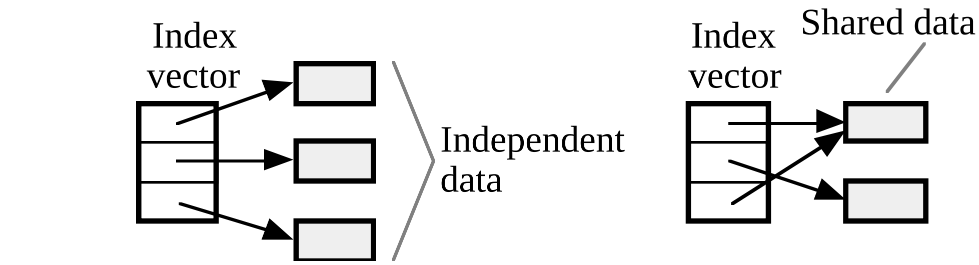
Figures



(a) A vector of pointers

(b) A vector of subscripts

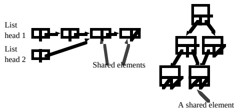
Figure 1. Two types of index vectors



(a) Unconditionally vector-processable data

(b) Read-only vector-processable data

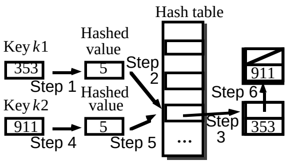
Figure 2. Vector-processable data for the previous methods



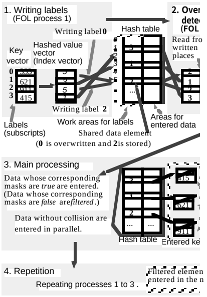
(a) Two lists with shared elements

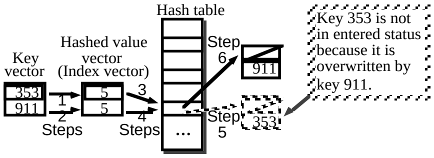
(b) A binary tree with a shared element

Figure 3. Examples of partially shared data structures



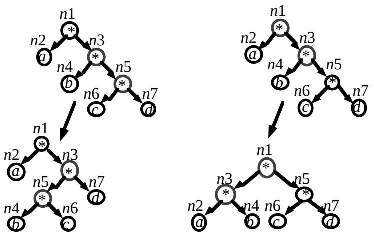
(a) Sequential processing





(b) “Forced” vector processing

Figure 4. The problem of multiple hashing by vector processing



(a) Possible rewriting 1 (b) Possible rewriting 2

Figure 5. Two ways of rewriting a tree

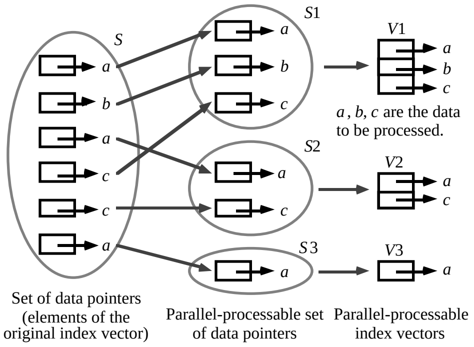


Figure 6. Decomposition of data with sharing into parallel-processable index vectors

Figure 7. A method of multiple hashing by FOL

```
input    table :       The hash table.
         key[1 : n] :  A set of keys to be entered {only keys are
                       entered in this algorithm}.
output   table :       The hash table where key[1 : n] are entered in.

local    hashedValue[1 : n], entered[1 : n].  /* local variables */

/* Computing hashed values and entering data into the table */
hashedValue[1 : n] := hash(key[1 : n]);
                /* Calculate hashed values     */
                /* {for example, hash(x) = x mod size(table)}. */
where table[hashedValue[1 : n]] = unentered do
                /* Detection of conflict among the data to be entered and */
                /* already entered data. */
        table[hashedValue[1 : n]] := key[1 : n];                        (2.1)
                /* Enter the keys only where they have not yet been entered. */
                /* More than one data item may be written into a hash table entry. */
end where;

for i in 1 .. size(table) loop
/* Checking unentered elements and collecting them */
        entered[1 : n] := (key[1 : n] = table[hashedValue[1 : n]]);
        nrest := countTrue(entered[1 : n]);                             (2.2)
                /* Count number of trues in boolean array 'entered'. */
        hashedValue[1 : nrest] := hashedValue[1 : n] where not entered[1 : n];
                /* Pack unentered elements in hashedValue[1 : n]. */
        key[1 : nrest] := key[1 : n] where not entered[1 : n];          (2.3)
                /* Pack unentered elements in key[1 : n]. */

/* Testing whether data entry is finished */
        if nrest = 0 then exit loop;     /* exit for-loop */
        n := nrest;

/* Computing the subscripts for the next step and entering data */
        hashedValue[1 : n] :=
                (hashedValue[1 : n] + (key[1 : n] & 31) + 1) mod size(table);
        where table[hashedValue[1 : n]] = unentered do
                table[hashedValue[1 : n]] := key[1 : n];                (2.4)
                        /* Enter the keys only where they are not yet entered. More  */
                        /* than one data item may be written into a hash table entry. */
        end where;
end loop;
```

Figure 8. Vectorized algorithm for entering data into a hash table

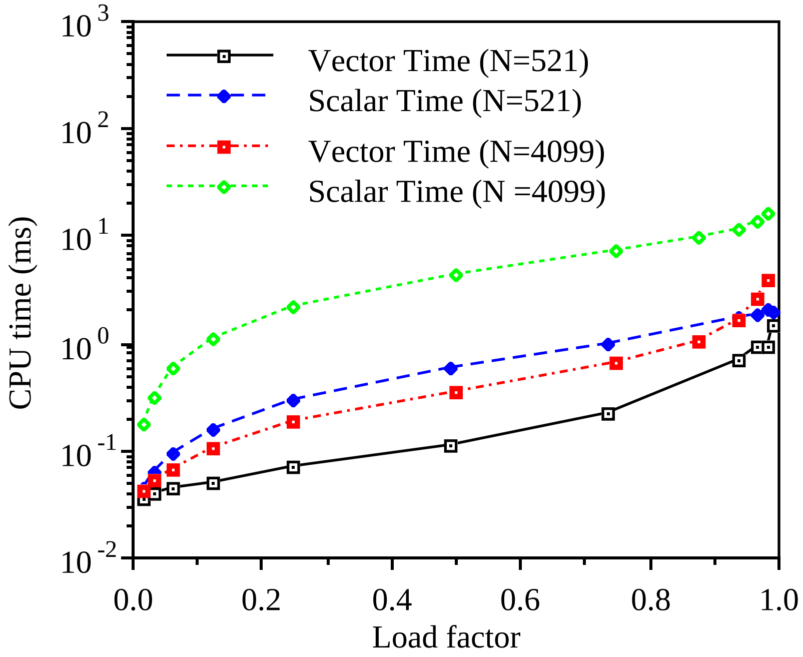


Figure 9. CPU time of multiple hashing into an empty hash table by S-810 (*N* : table size)

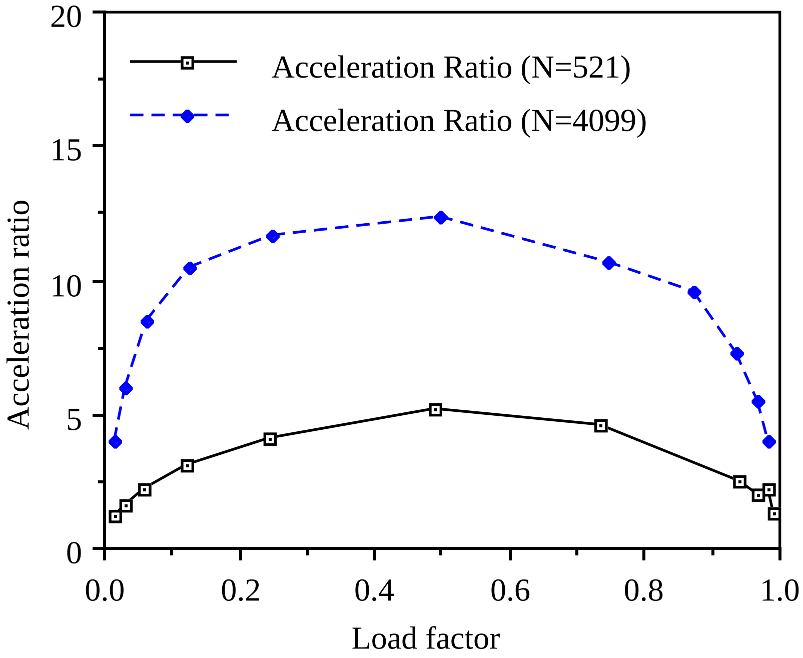


Figure 10. Acceleration ratio of multiple hashing into an empty hash table by S-810

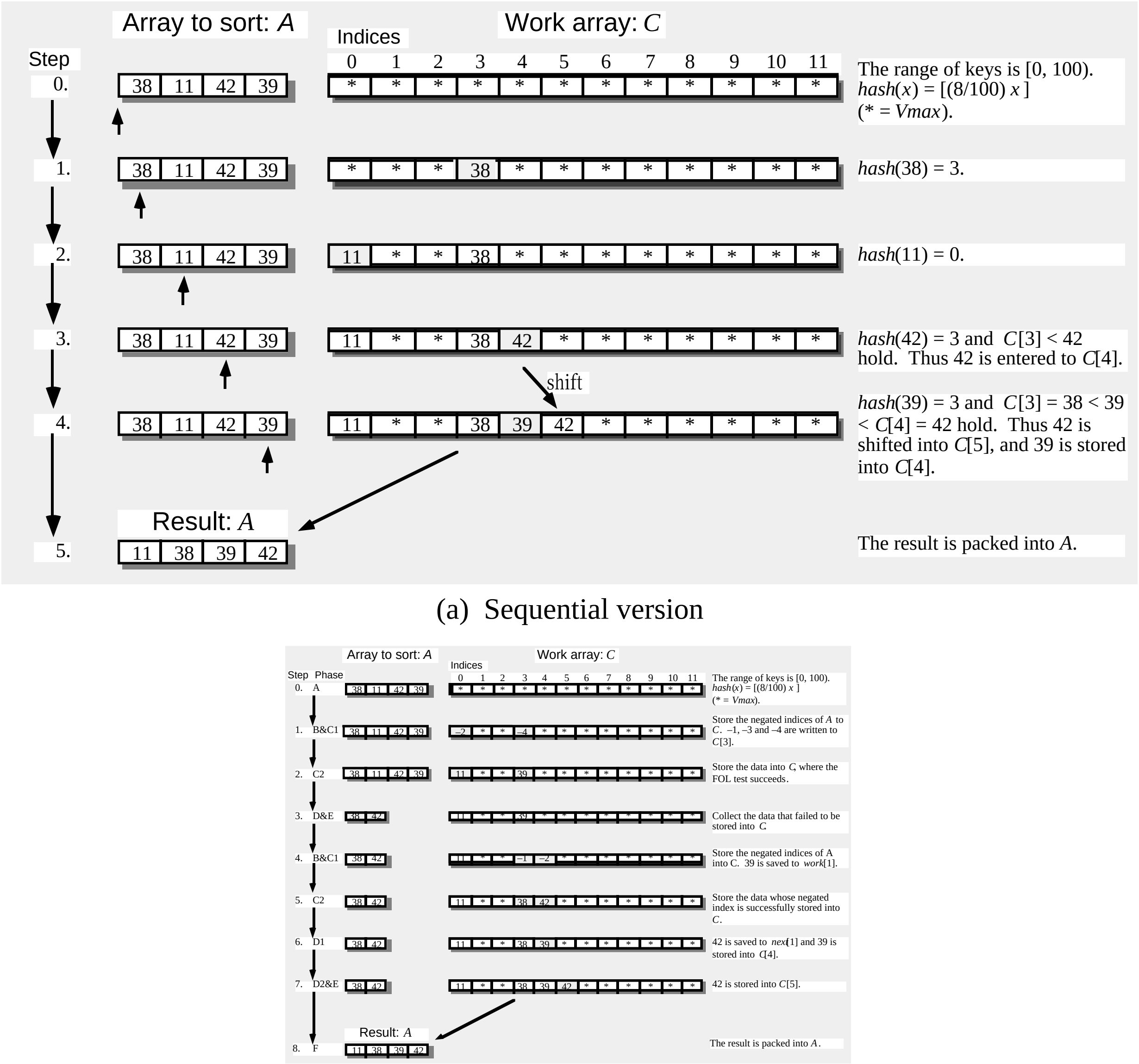
```
input    A[1 : n]:  Array to sort {the element values should be in [0, Vmax)}.
output   A[1 : n]:  Sorted array.

local    C[0 : 3*n – 1];

for i in 0 .. size(C) – 1 loop C[i] := unentered;  end loop;    /* Initialize C. */

/* Scatter the data into C: */
for i in 1 .. n loop
/* A. Computing a “hashed” value of A[i]. */
    hashedValue := int(float(2 * size(C) * A[i]) / Vmax);

/* B. Finding the table entry to insert new data A[i]: */
    while $C[hashedValue] \le A[i]$ loop
        hashedValue := hashedValue + 1;
    end while;

/* C&D. Inserting new data and shifting the data in C: */
    w := C[hashedValue];  C[hashedValue] := A[i];
    while $w \ne unentered$ loop
        hashedValue := hashedValue + 1;
        x := C[hashedValue];  C[hashedValue] := w;  w := x;
    end while;
end for;

/* F. Packing the sorted data into A. */
count := 0;
for i in 0 .. size(C) – 1 loop
    if $C[i] \ne unentered$ then
        count := count + 1;  A[count] := C[i];
    end if;
end for;
```

Figure 11. Sequential algorithm of the address calculation sorting

```
input   A[1 : n]:  Array to sort {the element values should be in [0, Vmax)}.
output  A[1 : n]:  Sorted array.

local   C[0 : 3*n – 1], uninsertable[1 : n], work[1 : n], entered[1 : n],
        toShift[1 : n], index[1 : n], next[1 : n], nonempty[1 : n].

C[0 : size(C) – 1] := unentered;        /* initialize C  (unentered = Vmax) */

/* A. Computing “hashed” values. */
hashedValue[1 : n] := int(float(2 * size(C) * A[i]) / Vmax);  nrest := n;

repeat
 /* B. Finding table entries to insert data. */
    repeat
        uninsertable[1 : nrest] := (C[hashedValue[1 : nrest]] $\le$ A[1 : nrest]);
            /* Check the first type of collision with stored data. */
            /* If hashedValue[i] $\neq$ unentered, the right-hand side */
            /* condition holds, i.e., there is a first type of collision. */
        Nuninsertable := countTrue(uninsertable[1 : nrest]);
            /* Count the number of uninsertable (colliding) data items. */
        where uninsertable[1 : nrest] do
            hashedValue[1 : nrest] := hashedValue[1 : nrest] + 1;
        end where;
    until Nuninsertable = 0;        /* Repeat until there is no first type of collision. */

/* C. Inserting the data. */
    work[1 : nrest] := C[hashedValue[1 : nrest]];
        /* Save the original values of C to work. */
    C[hashedValue[1 : nrest]] := $-\iota$;
        /* Store the identifiers to check {$-\iota$ is array (–1, –2, …, –nrest)}. */
        /* An entry of C may be written twice or more (overwritten). */
    entered[1 : nrest] := C[hashedValue[1 : nrest]] = $-\iota$;
        /* Check the second type of collision between newly entered data items. */
    where entered[1 : nrest] do
        C[hashedValue[1 : nrest]] := A[1 : nrest];      /* enter */
    end where;

/* D. Shifting the work array elements {only for successfully inserted data}. */
    toShift[1 : nrest] := entered[1 : nrest] and (work[1 : nrest] $\neq$ unentered);
    NtoShift := countTrue(toShift[1 : nrest]);
    work[1 : NtoShift] := work[1 : nrest] where toShift[1 : nrest];
    index[1 : NtoShift] := (hashedValue[1 : nrest] + 1) where toShift[1 : nrest];
    while NtoShift > 0 do
        next[1 : NtoShift] := C[index[1 : NtoShift]];
        C[index[1 : NtoShift]] := work[1 : NtoShift];
```

```
                nonempty[1 : NtoShift] := (next[1 : NtoShift] $\le$ unentered;
                count  := countTrue(nonempty[1 : NtoShift]);
                work[1 : count] := next[1 : NtoShift]
                                where nonempty[1 : NtoShift];        /* Pack work. */
                index[1 : count] := index[1 : NtoShift] + 1
                                where nonempty[1 : NtoShift];        /* Pack index. */
                NtoShift := count;
        end while;

/* E. Collecting not yet inserted data for the next iteration. */
        irest := countTrue(not entered);
        hashedValue[1 : irest] := hashedValue[1 : nrest] where not entered[1 : nrest];
        A[1 : irest] := A[1 : nrest] where not entered[1 : nrest];
        nrest  := irest;
until nrest = 0;          /* until all the data are inserted */

/* F. Packing the sorted data into A. */
A[1 : n] := C[0 : size(C) – 1] where (C[0 : size(C) – 1] $\neq$ unentered);
```

Figure 12. Vectorized algorithm of the address calculation sorting



The range of keys is [0, 100). *hash*(*x*) = [(8/100) *x*] (* = *Vmax*).

hash(38) = 3.

hash(11) = 0.

hash(42) = 3 and *C*[3] < 42 hold. Thus 42 is entered to *C*[4].

hash(39) = 3 and *C*[3] = 38 < 39 < *C*[4] = 42 hold. Thus 42 is shifted into *C*[5], and 39 is stored into *C*[4].

The result is packed into *A*.

(a) Sequential version

Array to sort: A
Work array: C
Indices
Step Phase
0 1 2 3 4 5 6 7 8 9 10 11
0. A 38 11 42 39 | * * * * * * * * * * * *
1. B&C1 38 11 42 39 | –2 * * –4 * * * * * * * *
2. C2 38 11 42 39 | 11 * * 39 * * * * * * * *
3. D&E 38 42 | 11 * * 39 * * * * * * * *
4. B&C1 38 42 | 11 * * –1 –2 * * * * * * *
5. C2 38 42 | 11 * * 38 42 * * * * * * *
6. D1 38 42 | 11 * * 38 39 * * * * * * *
7. D2&E 38 42 | 11 * * 38 39 42 * * * * * *
Result: A
8. F 11 38 39 42

The range of keys is [0, 100). *hash*(*x*) = [(8/100) *x*] (* = *Vmax*).

Store the negated indices of *A* to *C*. –1, –3 and –4 are written to *C*[3].

Store the data into *C*, where the FOL test succeeds.

Collect the data that failed to be stored into *C*.

Store the negated indices of A into C. 39 is saved to *work*[1].

Store the data whose negated index is successfully stored into *C*.

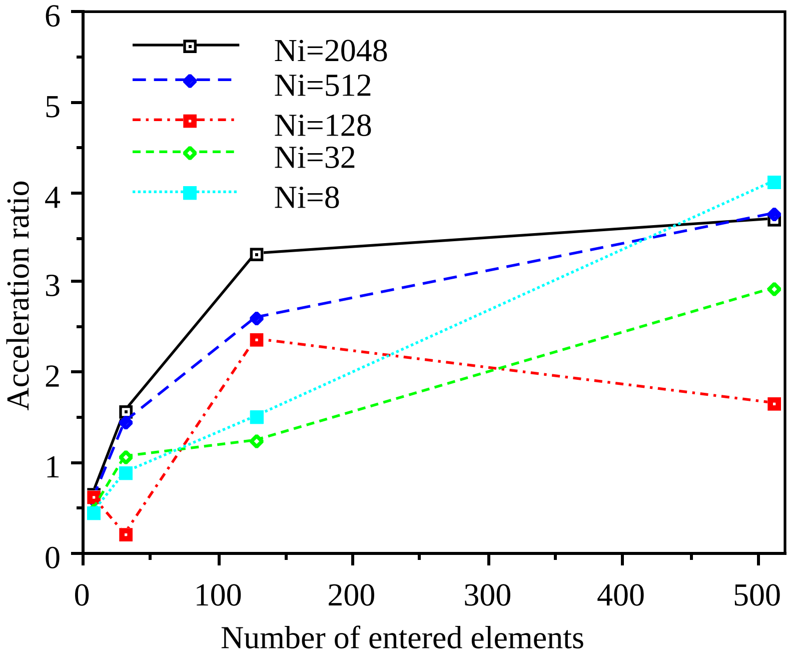
42 is saved to *next*[1] and 39 is stored into *C*[4].

42 is stored into *C*[5].

The result is packed into *A*.

(b) Vectorized version

Figure 13. Example of sequential and vectorized address calculation sorting



(*Ni* : table siz)

Figure 14. Acceleration ratio when entering multiple data items into a binary tree by S-810

Table

Table 1. CPU time and the acceleration ratio of $O(N)$ sorting algorithms

Algorithm	N	S-810/20 CPU time (μs)		Acceleration
		Sequential	Vectorized	ratio
Address Calculation Sorting*	2^6	289	110	2.62
	2^{10}	4,286	560	7.65
	2^{14}	66,955	5,215	12.84
Distribution Counting Sort**	2^6	12,206	1,522	8.02
	2^{10}	13,072	1,738	7.52
	2^{14}	30,089	5,667	5.31

* The size of work array C is $3n$.
** The size of work array is 2^{16}, which is the range of the data.

Footnotes

[*] This research was done at Central Research Laboratory, Hitachi Ltd.

[1] The hardware must be capable of serializing vector operations, for FOL to be applied. Most vector processors and data-parallel processors have this function.

[2] FOL is a generalization of the "overwrite-and-check" method in Kanada [8].

[3] The concrete example shown in Section 4.1 may be helpful for understanding the principle of FOL.

[4] This condition holds for multiple hashing as the order does not matter when there are no collisions. If a collision occurs, the order of the data entered in the hash table is dependent on which colliding data is stored, but once again this is not important. Processing not satisfying this condition is considered in a later footnote.

[5] For example, the execution order may affect the order of entered data in the case of multiple hashing. In Figure 4 (a), 911 may be at the top of the chain, instead of 353. However, this does not affect the correctness.

[6] The most easily computable label for element v in vector V is the index or element number of v in V, or the displacement of v (the number of bytes) from the top address of V.

[7] There is a type of algorithm, which FOL should be applied to, in which the order of multiple processings for *one* data item must be preserved. It is possible to modify FOL1 so as to eliminate the above process condition and to construct vectorized algorithms of such a type. That is, if processing P_i is applied to data item d_i (= $v_i \rightarrow d$) and processing P_j is applied to data item d_j (= $v_j \rightarrow d$) where d_i and d_j are equivalent (i.e., in the same storage area), and the execution of P_i precedes that of P_j in the sequential execution, then the ELS condition can be replaced by a stronger condition so that the relation $k > l$ holds for the output sets S_k and S_l where $d_i \in S_k$ and $d_j \in S_l$.

For vector processor S-3800, a higher-performance list-vector store instruction, the `VIST` (Vector Indirect STore) instruction, which satisfies the ELS condition, can be used for writing labels in FOL1. However, the `VSTX` (Vector STore indeXed) instruction, which is slower but guarantees the order in which vector elements are stored (a stronger condition than the ELS condition), can be used instead of the `VIST` to satisfy the above relation, $k > l$, and to eliminate the above condition.

[8] A deadlock may occur in Step 4. A solution to this problem will be explained later.

[9] The acceleration ratio means the ratio of the vectorized total execution time and the *original* sequential execution time.

[10] However, because the vectorized algorithm cannot be written in standard Fortran, the program violates its semantics using the vectorization forcing option (`*VOPTION`).