D

Implementing the Multiprefix Operation on Parallel and Vector Computers

Thomas J. Sheffler

August 18, 1992

CMU-CS-92-173

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

423887

26P8

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. Cray Y-MP time was provided by the Pittsburgh Supercomputing Center (Grant ASC890018P).

Keywords: Data-Parallel, parallel algorithms, supercomputers, vectorized algorithms, sorting, multiprefix, sparse-matrix vector multiplication

Abstract

For an ordered set of n values, each with an associated integer label, the multiprefix operation calculates a partial sum for each value that is the sum of all preceding values with the same label. The multiprefix operation has been proposed as a parallel primitive because of its power for expressing many data parallel algorithms succinctly. However, most approaches to implementing this operation have used integer sorting to gather elements with the same label together or have suggested special hardware.

In this paper we present a work efficient algorithm for the multiprefix operation on n elements that runs in $S = O(\sqrt{n})$ parallel steps on a $p = \sqrt{n}$ processor CRCW-ARB PRAM. The CRCW-ARB model ensures only that of multiple processors writing to the same location, an arbitrary one succeeds. We make use of this feature to resolve data dependencies in the first phase of the algorithm only so that all later steps guarantee EREW memory access.

A fully vectorized version of our algorithm has been designed for the CRAY Y-MP and provides good performance for a number of important algorithms. For the integer sorting test of the NAS benchmarks, our multiprefix operation was used to create an algorithm that is competitive in performance with the current best algorithms for that machine. As another example, we show that by using the multiprefix operator for sparse-matrix dense-vector multiplication, we obtain performance exceding traditional FORTRAN-based approaches. Finally, our algorithm also makes possible the simultion of a CRCW-PLUS PRAM on a p processor CRCW-ARB PRAM with only constant slowdown for problem sizes $n > p^2$.

1 Introduction

Each of an ordered set of data values $A = \langle a_1, a_2, ..., a_n \rangle$ are associated with integer labels $L = \langle l_1, l_2, ..., l_n \rangle$ with $l_i \in \{1, 2, ..., m\}$. The multiprefix problem is to compute a collection of partial sums $S = \langle s_1, s_2, ..., s_n \rangle$ for the values and a set of reductions $R = \{r_1, r_2, .., .r_m\}$ for the labels such that

$$s_i = \sum \{a_j \mid l_j = l_i \text{ and } j < i\}$$

and

$$r_k = \sum \{a_j \mid l_j = k \text{ and } k \in L\}.$$

That is, for each value its multiprefix result is the sum of all elements with its same label preceeding it in vector order. And for each label, its reduction is the sum of all values with that label. For example, given a vector of values, A, and labels, L, as shown in Figure 1, a call to the multiprefix operator would produce the results in S and R as illustrated. Because only preceding values contribute to each sum, the first sum of each "group" of equal labels is 0, and because only the labels 2 and 3 appear in the L vector, the reduction vector R has non-zero values only at these positions.

The general multiprefix operator solves the multiprefix problem and extends the summing operation to any binary associative operator on values of arbitrary type. As an operator, it accepts vectors containing the values and labels, a binary associative operator, and computes the values of the sums and reductions with respect to the operator given. Typical operators are MAX, MIN, PLUS, MULT, AND and OR on data types INTEGER, FLOATING and BOOLEAN. Throughout this paper we will concentrate on the multiprefix-PLUS operator, but our discussion generalizes to any binary associative operator as long as 0 is replaced with the appropriate identity element for the operator chosen.

The multiprefix operator has been previously proposed as a parallel primitive for the Fluent abstract machine [RBJ88] and as a general purpose parallel primitive [Coh90]. Our definition of the multiprefix operator is nearly identical to the one given in [Coh90] but differs from the one given in [RBJ88] slightly. In that formulation, the labels were references to shared variables to which the reduction values were written, and the operation was not presented in a data parallel framework. These differences are inconsequential.

The multiprefix operator subsumes the functionality of many other parallel primitives. For instance, it iprovides the functionality of the fetch-and-op primitive of the NYU Ultracomputer [GLR81]. While the fetch-and-op primitive is non-deterministic in its evaluation order, the multiprefix operator ensures that results are computed in vector index order. Multiprefix also provides the functionality of the segmented-scans [Ble90] and the combining-sends of the Connection Machine [Hil85], and can be used to implement the β operation of CM-Lisp [SH86]. A segmented-scan is simulated by distributing the same label to each element in a segment and then executing the multiprefix operation. A combining-send operation is provided directly by multiprefix, but only the reduction values are used. When the multiprefix sums are not computed, we call this a "multireduce" operation. The multireduce operation occurs most frequently as histogram computation which is important enough that a special "Vector Update Loop" compiler directive has been suggested to identify this procedure [PMM92, page 18]. In short, the multiprefix operation attempts to unify the functionality of these many varied parallel primitives.

```
A      = [ 1  1  5  1  6  1  7  ]
L      = [ 3  3  2  3  2  3  2  ]

MP(A, L, +, S, R);

INDEX =    1  2  3  4  5  6  7
S      = [ 0  1  0  2  5  3  11 ]
R      = [ 0   18 4  ]
```

Figure 1: An example multiprefix result given a vector of data values (A) and an associated vector of labels (L). The two results computed are the multiprefix sums (S) and the reductions for each label (R). Each reduction value represents the sum of all values with that label.

Table 1: Comparison of Integer Sorting Algorithms on the CRAY Y-MP for the NAS Integer Sorting benchmark. This test involves sorting 8 million 19-bit integers 10 times.

NAS Integer Sorting Benchmark	
Method	Time (Secs)
Partially Vectorized FORTRAN Bucket Sort	18.24
Cray Research Inc. Implementation	14.00
Our Multiprefix-based Sort	13.66

1.1 Applications of Multiprefix

Multiprefix has been proposed as a parallel primitive because of its generality and the power that it provides for expressing many parallel algorithms. For example, [RBJ88] show how to implement an integer sorting routine using multiprefix in just a few steps. We also show that multiprefix can be used to efficiently implement sparse-matrix vector multiplication.

A vector computer with scatter/gather capability may simulate a synchronous PRAM algorithm by issuing one vector operation for each parallel step [CBZ90]. Using this approach, we implemented a fully vectorized version of the multiprefix operator on the CRAY Y-MP and constructed a fast integer sorting algorithm that is competive in performance with the current best approaches. The NAS parallel benchmark suite is a collection of 8 test problems intended to be used to compare parallel machines [BBCS91]. The "Integer Sorting" benchmark requires the sorting of 8 million 19-bit integers. Table 1 shows a comparison of the performance times reported for three different approaches on the CRAY Y-MP at the time of writing this paper [BBB+91]. While the time reported for our multiprefix approach represents a very young implementation with little optimization, it still outperformed the other two approaches.

Our vectorized multiprefix operator was also used to implement a sparse-matrix by dense-vector multiplication routine on the CRAY Y-MP. This mathematical kernel is very important in many numerical applications. The two approaches compared to ours are a traditional row-major algorithm that uses "compressed sparse row" storage, and an algorithm based on a "jagged-diagonal" storage format [Saa89]. Of these, the jagged-diagonal approach trades off a large preprocessing time for enhanced vectorization of the numerical portion of the algorithm. The row-major algorithm suffers

Table 2: A comparison of the performance sparse-matrix vector multiplication routines on the CRAY Y-MP. The size indicates the dimension of each matrix, while ρ indicates the density. For very large, sparse matrices, the multiprefix approach excels, while the other methods are better suited to matrices of greater density.

Sparse Matrix-Vector Multiplication (Times in mS)				
Order	ρ	Compressed-Row	Jagged-Diag	Multiprefix
15000	0.001	30.29	28.09	27.43
10000	0.001	19.52	16.31	12.43
5000	0.001	9.48	6.99	3.45
2000	0.005	3.90	3.23	2.77
1000	0.010	1.95	1.66	1.50
100	0.400	0.27	0.42	0.76

from poor vectorization because of the very short rows for sparse systems. Table 2 summarizes our results for a number of matrices of varying size and density (ρ) and shows that the multiprefix approach to this problem is competitive to more traditional FORTRAN-based approaches.

1.2 Theoretical Results

In this paper we present an algorithm for implementing the multiprefix operator for n values in $S = O(\sqrt{n})$ parallel steps on a $p = \sqrt{n}$ processor CRCW-ARB PRAM in $s = O(n + m)$ space. Another important complexity measure of an algorithm is the total work performed and is the sum over all steps of the number of primitive, scalar instructions issued. For a serial algorithm on one processor this is its traditional time complexity. A parallel algorithm is *work efficient* if it performs no more work than an equivalent serial algorithm. Our multiprefix algorithm performs $W = O(n)$ work; hence it is work efficient.

Of the CRCW PRAM models, the CRCW-ARB model assumes only that of multiple processors writing to the same location, an arbitrary one succeeds. While not as restrictive as the EREW model, the CRCW-ARB PRAM model is a fairly realistic representation of many current parallel architectures that provide a shared memory. The CRCW-PLUS PRAM model allows a combining function to be applied to values concurrently written to the same location [CLR89, page 690]. Our multiprefix algorithm can be used to simulate a concurrent combining write for problem sizes $n \geq p^2$. Consider a parallel algorithm for a problem of size $n = \alpha^2 p^2$ ($\alpha \geq 1$) on a p-processor CRCW-ARB PRAM. With each processor simulating α virtual processors in $O(\alpha)$ steps, our algorithm may be used to simulate a concurrent combining write in $O(\alpha p)$ virtual parallel steps, or $O(\alpha^2 p)$ real parallel steps. Any other algorithm for a problem of the same size would require $O(\alpha^2 p)$ steps as well, so that our simulation is optimal when $n \geq p^2$. Using our algorithm, we are able to claim that for p processors:

- A CRCW-PLUS PRAM may be simulated on a CRCW-ARB PRAM with only constant slowdown for problem sizes $n \geq p^2$.

Our algorithm makes use of two novel techniques: an "overwrite-and-test" memory access method and the creation of a "spinetree" data structure. The overwrite-and-test" memory access method is an arbitration scheme related to parallel hashing [Kan90]. With this technique, all processors vying for a particular resource send a unique value to a memory location assigned to

```
labels:  int[n];
values:  int[n];
buckets: int[m];
multi:   int[n];

SERIAL-INITIALIZATION:
    for (i = 1 to n)
        buckets[label[i]] = 0;


SERIAL-MULTIPREFIX:
    for (i = 1 to n) {
        multi[i] = buckets[label[i]];
        buckets[label[i]] += value[i];
    }
```

Figure 2: A simple serial algorithm for the multiprefix operation. This algorithm exhibits works properly only if the elements are processed in order, making it unsuitable for parallel implementation.

that resource. If an attempt is made to write many values to the same location, the arbitrary value written identifies the processor that wins the resource. In our multiprefix algorithm we use this technique to build a special tree data structure called a "spinetree" that represents a virtual combining network. By performing these operations only in the first phase of our multiprefix algorithm, all remaining phases proceed with guaranteed EREW memory access.

2 Implementing Multiprefix

The multiprefix operation is most easily described by a straightforward serial algorithm. After introducing the serial approach, we will explain why it is difficult to parallelize and then introduce our parallel algorithm.

2.1 Serial Multiprefix

A simple serial algorithm for the multiprefix operation is shown in Figure 2. The n values are stored in a vector called *values* with a corresponding label in the vector called *labels*. The vector *multi* will hold the multiprefix values computed. The labels are known to lie in the range $[1, 2, ..., m]$. A temporary vector called *buckets* is allocated to be as large as m to hold the reduction values for each of the labels. Because the labels are integers no greater than m, they directly index sites in the bucket vector. The initialization step clears all needed memory locations, but avoids accessing all of the bucket entries by only clearing those sites referenced by the labels.

The main loop simply processes the elements in order. At each step i, the current value of the bucket referenced by *label[i]* becomes the multiprefix value *multi[i]*. Then, the bucket is incremented by the appropriate amount using the increment operator (+=). This loop is similar to the main procedure of a bucket sort, or a general histogramming operation for integer keys, except that those procedures do not save the value of the bucket before incrementing it. These intermediate values

```
type spinerec = record {
    rowsum: int;
    spinesum: int;
    multisum: int;
    spine: ptr to spinerec;
};

bucket:  spinerec[m];
temp:    spinerec[n];

label:   int[n];
value:   int[n];
multi:   int[n];

INITIALIZE:
    pardo ($i$ = 1 to $n$) {
        temp[i].rowsum = 0;
        temp[i].spine = &bucket[label[i]];
        bucket[label[i]].spine = &bucket[label[i]];
        bucket[label[i]].rowsum = 0;
    }
```

Figure 3: The type definition and initialization phase for the parallel multiprefix algorithm. All temporary storage is cleared or set in one parallel step.

of the buckets *are* the multiprefix sums. By extending this operation to arbitrary data types with an arbitrary summing function this algorithm implements a general multiprefix operator.

This loop is difficult to parallelize. It is clear that all elements may not be processed in parallel because elements with the same label would conflict in their modification of the same bucket. Even if accesses to only the same bucket could be serialized a total of n steps would still be required in the worst case when all labels are the same,

2.2 Parallel Multiprefix

Our parallel algorithm for the multiprefix operation is shown in Figures 3 and 4. Figure 3 describes the record structure used for temporary values while Figure 4 describes the four phases of the algorithm. As before, the n values and their labels are stored in vectors *value* and *label*. The multiprefix sums are written to *multi*, and the reductions are left in the temporary vector called *bucket*, of size m.

The *spinerec* record type is used to store the temporary information associated with each bucket and value/label pair in the algorithm. (We will often call a value/label pair an "element" when describing the algorithm.) The *spine* field is a pointer that connects the elements and buckets into a structure called a "spinetree." The other integer fields will be explained along with the later phases of the algorithm.

In the initialization phase, the temporary storage of each element and the bucket that each element references is cleared as before. The *spine* pointer of each element is set to the address of its bucket using the address-of operator (&), and the *spine* pointer of each bucket is set to itself.

```
SPINETREE:
    for ($r = \sqrt{n}$ downto 1)
        pardo ($i = ((r-1)\sqrt{n}+1)$ to $(r\sqrt{n})$) {
            temp[i].spine = bucket[label[i]].spine;
            bucket[label[i]].spine = &temp[i];
        }

ROWSUMS:
    for (c = 1 to $\sqrt{n}$)
        pardo ($i = c$ to $n$ by $\sqrt{n}$)
            with temp[i] do
                spine → rowsum += value[i];

SPINESUMS:
    for (r = 1 to $\sqrt{n}$)
        pardo ($i = ((r-1)\sqrt{n}+1)$ to $(r\sqrt{n})$)
            with temp[i] do
                if (rowsum $\neq$ 0) then
                    spine → spinesum = spinesum + rowsum;

MULTISUMS:
    for (c = 1 to $\sqrt{n}$)
        pardo ($i = c$ to $n$ by $\sqrt{n}$)
            with temp[i] do {
                multi[i] = spine → spinesum;
                spine → spinesum += value[i];
            }
```

Figure 4: The body of the parallel multiprefix algorithm is executed in four main phases. The SPINETREE phase builds the tree of data values. The other three remaining phases then execute with no memory access conflicts.

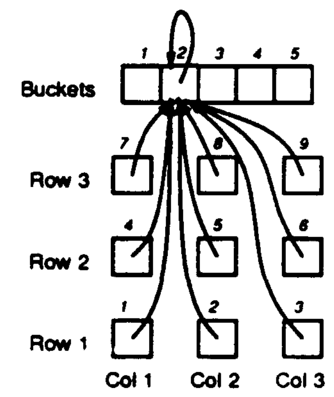

Figure 5: The pointer structure of the spinetree for 9 elements, each with the label 2. Because only the buckets actually used are initialized, only bucket number 2 is set to point to itself; the other buckets have unknown values in their pointer field.

These operations may be performed in parallel for all elements using concurrent writes and reads of the buckets.

The four main phases of this algorithm will be explained using an example involving 9 elements, all of which have the label 2 and a value 1. The result of this initialization step is the structure shown in Figure 5. All elements direct their pointers to bucket number 2 and the buckets are set to point to themselves.

The 9 elements are shown numbered in vector order but are conceptually arranged into a square. This row and column arrangement is important because later phases will operate in parallel on all of the elements in entire rows or columns in a **pardo** statement. Loops that access rows use a control variable r while loops over the columns use a c. Rows are numbered from bottom to top, and columns from left to right. Given a row r, the elements on that row lie in the range $[((r-1)\sqrt{n}) + 1, \ldots, r\sqrt{n}]$. Similary, the elements of column c are given by the sequence $[c, c + \sqrt{n}, c + 2\sqrt{n}, \ldots c + (\sqrt{n} - 1)\sqrt{n}]$. These formulas involve simple array address calculations.

This arrangement into rows and columns requires that n be a square. When this is not the case, it is a simple matter to pad the elements up to a square. Later, we will show how this can be avoided in certain circumstances.

The SPINETREE phase links the elements together into the spinetree structure. For each row, from top to bottom, the ***spine*** value of each element is replaced with that of its bucket using a concurrent read. Then, by using a concurrent write, the spine pointers of the buckets with elements on the currently active row are overwritten by the address of one of these elements.

This process is illustrated in Figure 6 with the initial state of the pointers shown on the left, and the pointer configuration after each row update. The active row is highlighted for each step. After the top row executes this step only the bucket pointer is changed to point to one of the elements of the top row. When the middle row is updated, each element is set to reference an arbitrary element with the same label in the preceding row. Finally, each of the elements of the bottom row readjust their pointers so that they now also reference an element with the same label in the preceding (middle) row.

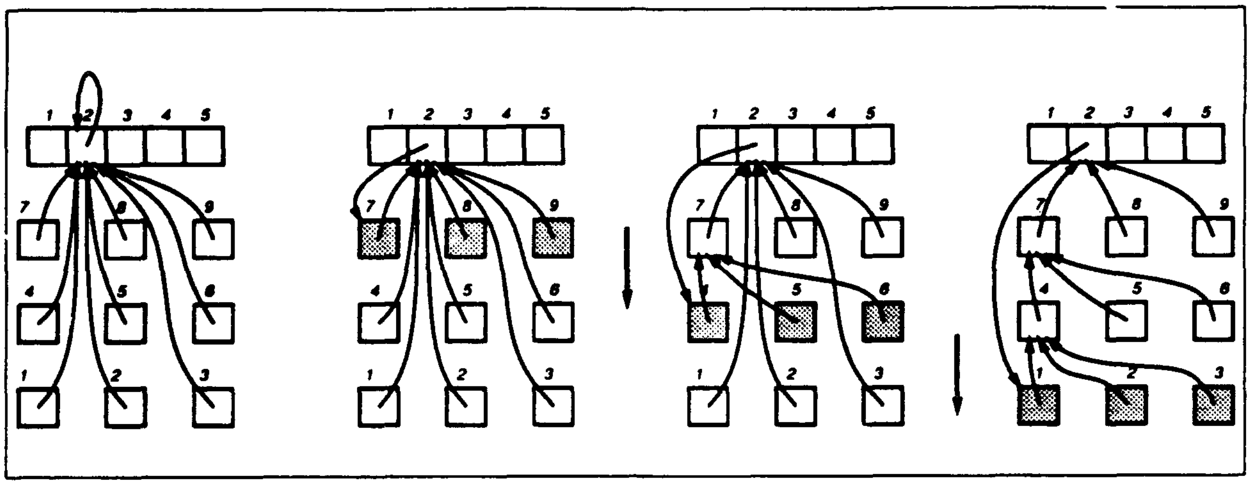

Figure 6: The evolution of the spine pointers during the SPINETREE phase of the algorithm. The rows are processed in order with the elements of each first reading the pointer of its bucket, and then attempting to write their own address there. At the end of this process elements with the same label are linked into the spinetree data structure.

The pointers now describe a tree with the property that every element is either on a special path called the "spine" or points directly at an element on the spine. In this tree, each child has a pointer to its parent. A "spine element" is defined as any that has a child in the tree, and the spine is the path that connects the spine elements. In our example, the spine includes elements 4 and 7 and the bucket. (The pointer from the bucket is no longer used and is not considered part of the tree.) Because all elements have the same label (2), they form a connected tree. In general, when there are different labels, each set of elements with the same label (called a "class") forms its own spinetree with its bucket as the root.

The SPINETREE phase uses an "overwrite-and-test" memory access method to determine which elements of a class will be the spine elements. The elements of each row attempt to "overwrite" their bucket to become a spine element. Elements of the next row "test" the bucket pointer value to determine the element of the preceding row that becomes their parent. In this manner, a tree is built such there is only one element of each class on each row that has children. The other three phases use this spinetree data structure to ensure that only one element ever attempts to update its parent in parallel, ensuring EREW memory access.

Snapshots of the intermediate values for each of the elements after the remaining three phases are shown in Figure 7. An arrow indicates the order of access by rows or columns. The ROWSUMS phase sweeps across the columns and operates on all elements in a column in parallel. Because the parent of each element must be in a preceding row, elements with the same label in a column will have different parents. By accessing the elements in columns, each element increments the ***rowsum*** value of its parent without conflict. At the end of this phase, each spine element is left with the sum of its children in the field ***rowsum***. Elements that are not on the spine will have a 0 ***rowsum*** value.

In the SPINESUMS phase, a prefix sum is computed along each spine in the ***spinesum*** field. Working from the bottom row to the top, each spine element sends the sum of its ***spinesum*** and ***rowsum*** value to its parent, calculating a recurrence along the spine. Since only spine elements will have a non-zero ***rowsum*** value, the conditional ensures that only spine elements participate. Because there is only one spine path through each spinetree, there are no memory access conflicts

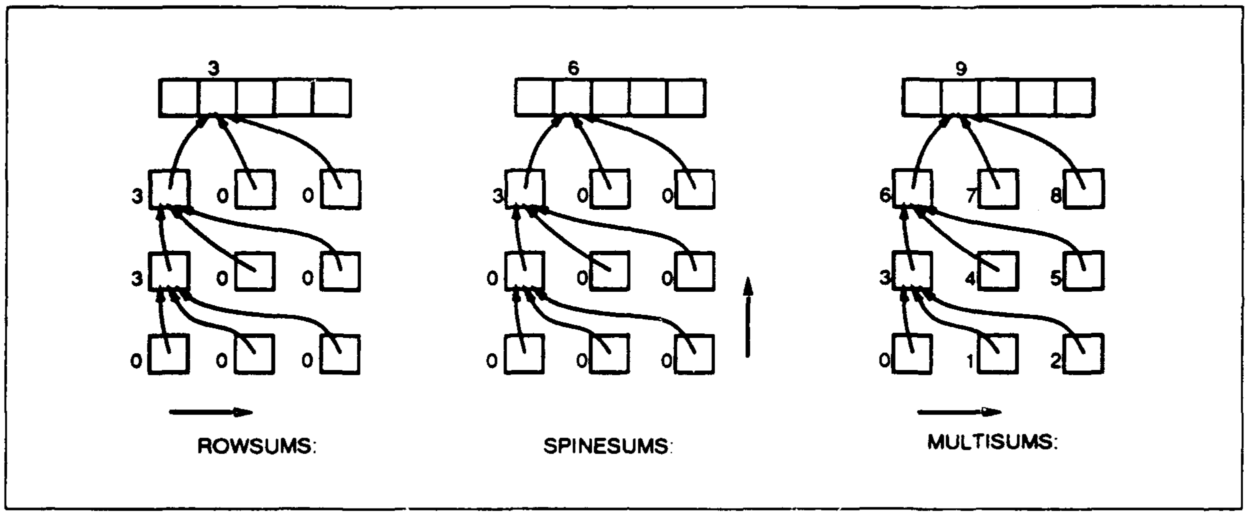

Figure 7: The pointer structure of the spinetree for 9 elements, each with the label 2. Because only the buckets actually used are initialized, only bucket number 2 is set to point to itself; the other buckets have unknown values in their pointer field.

as children update their parents. At the termination of this phase, each spine element will have in its ***spinesum*** field the sum of the elements in its class preceeding any of its children. Hence, each bucket will be left with the sum of all elements in its class not including those of the top row. The reduction value for each class may be calculated directly at this point by adding together the ***rowsum*** and ***spinesum*** values of the buckets.

In the last phase, called MULTISUMS, the final multiprefix values are distributed to each of the elements. Initially, each spine element has in its ***spinesum*** field the sum of all elements in its class preceeding any of its children. By accessing the columns in order, each child reads this value as the sum of all elements in its class preceding it, and then increments its parent for the next element of its class on the same row. Once again, access by columns ensures that no two children attempt to update the same parent concurrently. Because the columns are accessed in order, each child is guaranteed of receiving its multiprefix sum in vector order.

The final values shown in Figure 7 show the results of a simple multiprefix addition on a vector of 1's with the same label. As expected, the multiprefix operation serves to enumerate these values beginning at 0 and leaves a count of how many values there are in the bucket.

3 Algorithmic Analysis

The analysis of this algorithm is straightforward because the memory access patterns are very regular. We will make use of two measures to express the complexity of our algorithm. The first is the step complexity (S) which measures the total number of parallel steps required by the algorithm. This is the traditional parallel time measure expressed for most PRAM algorithms. The second is the work complexity (W) which measures the total number of elements operated on over all steps. A parallel algorithm that has a work complexity no greater than an equivalent serial algorithm by a constant factor is called "work efficient."

The initialization phase executes in a single parallel step. Each of the other four phases operates on an entire row or column in parallel with the **pardo** statement in the inner loop. The outer loop iterates over columns or rows for the inner parallel loop. Since there are $\sqrt{n}$ rows and columns, the outer loops of all four phases execute exactly $\sqrt{n}$ parallel steps. The parallel step complexity of the entire algorithm is $S = O(\sqrt{n})$.

The total amount of work performed by this algorithm is the sum over all steps of the number of all elements operated on. The initialization phase obviously performs $W = O(n)$ work. The later phases require $O(\sqrt{n})$ steps and operate on exactly $\sqrt{n}$ elements. Therefore, the work complexity of this algorithm is $W = O(n)$. Because a serial algorithm would also have to perform $O(n)$ work (by visiting all of the elements) this algorithm is work efficient.

3.1 Correctness

The algorithm depends on some special properties of the spinetree data structure. In this section we will discuss those properties and show that they are maintained for any possible labeling of the values.

Theorem 1 *Elements have the same parent iff they have the same label and are in the same row.*

Proof: For each bucket b, let $R_b = \langle r_k..r_1 \rangle$ be the ordered set of rows in which elements have the label b where $r_{i+1} > r_i$. All elements update only their own bucket, so we may consider each class independently. Since the SPINETREE procedure operates in reverse row order, we need only consider for each class b those steps involving rows in R_b in order.

All elements in row r_i with label b replace their spine pointer with that of their bucket and thus have the same parent. Bucket b is then overwritten with a pointer to a member of row r_i. Because the elements may be only a member of one row the pointer values overwritten for each row will be unique and elements will find the same parent only if they were processed together, on the same row.
□

Corollary 1 *The children of a spine element are in different columns.*

Proof: Since the children of a spine element are in the same row, none of them may be in the same column.
□

From these properties we can guarantee that when operating on elements in the same column in parallel, these elements may update their parent with no other member of their class interfering. Also, since each element's parent is in another row, when operating on all elements of the same row in parallel each may modify its parent with the assurance that their parent is not also active. In short, by using the arbitrary concurrent WRITE of the SPINETREE phase we have created a data structure in which EREW memory access is guaranteed for the remaining three phases of the algorithm.

Theorem 2 *There is at most one spine element per class per row.*

Proof: A spine element is one that has children. An element is only a candidate for having children if it manages to write its value into its bucket's ***spine*** field. This can only be done while its row is active. Because of the write ARB only one element of its class in its row may succeed to become a candidate for having children.
□

Corollary 2 *A spine element has at most one child that is also a spine element.*

Proof: From the preceding theorems. Two children of a spine element could only be spine elements if they were on different rows, and this is impossible since children of the same parent are necessarily on the same row.
□

This corollary ensures us that the spine path through the tree for each class has at most one spine element per row. This property is important for the proper functioning of the SPINESUMS phase of the algorithm. In that phase, the spine elements forward their value to their parents. If two spine elements could exist on the same row, or if a spine element could have two children that are spine elements, then there could be write conflicts. Since only spine elements may have a non-zero ***rowsum*** value, by processing the rows in order from bottom to top, the sums computed along the spines (the ***spinesum*** value) add together only the ***rowsum*** values of the spine elements.

4 Implementation on the CRAY Y-MP

We implemented our algorithm in C on the CRAY Y-MP. We chose C only because most of our other tools were written in C. This also allowed us to more easily use the C-preprocessor to provide all of the variants of multiprefix we desired with one main template source file. These included such variations as ADD, MULT, MAX, MIN, AND, OR on data types INTEGER, DOUBLE and BOOLEAN. Full vectorization of the algorithm was achieved by directing the compiler to vectorize each of the inner **pardo** loops. Because the vectorization of these loops is straightforward, it would also be simple to translate the algorithm to FORTRAN.

For our implementation on the CRAY Y-MP we modified the algorithm slightly to use array indexing instead of pointers. This required two simple changes. The first change was to arrange bucket memory to be contiguous with the prefix element memory. This was easy to ensure at the time of allocation. Temporary memory was allocated in one block and divided by a "pivot" point as shown in Figure 8. Memory to the left was reserved for buckets, and memory to the right for the elements. In this manner the buckets were addressed by small integers between 1 and m, and the elements were addressed by integers in the range $m+1$ to $m+n$. Because references to the elements were generally made with respect to a row or column, the conceptual arrangement of the temporary memory that of a square array with a "handle" for the buckets.

Access of the elements by rows or columns was complicated only by the additional offset. Access by rows, as in the SPINETREE and SPINESUMS phases, simply offset the loop variable i by m. The same change occurred for column access in the ROWSUMS and SPINESUMS phases. By carefully recoding these loops, the compiler was able to deduce that a simple offset was added, as if some references began at an unnamed array allocated at location $m+1$.

The second change required unpacking the fields of the ***spinerec*** record type so that each field was allocated as a separate vector. Instead of one vector of records with four fields, we used four arrays called ***spine***, ***rowsums***, ***spinesums*** and ***prefixsums***. Using this arrangement, the spinetree data structure could be described by a single vector of length $(n+m)$ of integers no larger than $(n+m)$. Figure 9 shows the spinetree structure of the earlier example in its pointer form, and in the integer vector form allowed when the elements are contiguous with the buckets. The vector index of the buckets and elements is shown above each element; it is these indices that the ***spine*** values indicate. This allowed the pointer dereferencing operations to be implemented as direct scatter/gather operations using array indexing.

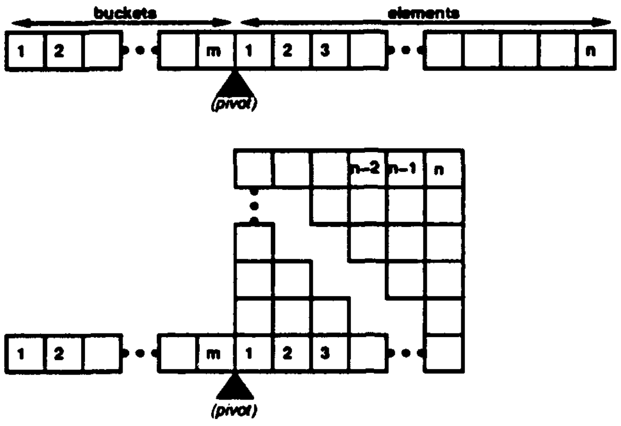

Figure 8: Temporary memory for the buckets and elements is allocated in a contiguous block but divided at the "pivot" point. Buckets are located at sites 1 through m, while the elements are offset at positions $m+1$ to $m+n$. The rows and columns of the elements are indicated by the conceptual arrangement shown.

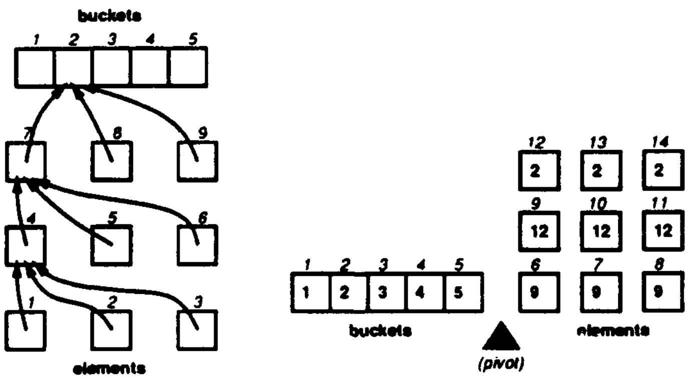

Figure 9: A single integer array is used to describe the spinetree data structure. On the left is shown the spinetree from the earlier example. On the right, is its equivalent representation with the index of the elements now offset by m. While the buckets and elements are allocated in one array, it is conceptually divided at the pivot point into the shape shown.

Table 3: Vector characterization parameters for the loops of each of the four phases of the multiprefix algorithm. The asymptotic time per element (t_e) is in 6nS clocks per element on the Y-MP. The short vector half-performance lengths indicate that these loops perform well for small problem sizes.

Vector Characteristic Parameters		
Phase	t_e (6nS clk/elt)	$n_{1/2}$
SPINETREE	5.3	20
ROWSUM	4.1	40
SPINESUM	7.4	20
PREFIXSUM	6.9	40

This arrangement also relieved a source of potential memory conflicts that could arise in the record based implementation when referencing the same field in continguous records. Since the record required 4 words of storage, any sequential access of the same field in the records would result in a memory access with stride 4. Such an access pattern would only make use of 1/4 of the memory banks available.

One last minor change was made to the initialization phase. In almost all applications the number of buckets, m, is no more than the number of elements, n. In the modified initialization we accessed each bucket directly rather than indirectly through the elements. While this altered the theoretical complexity measure for the real algorithm, in practice it was always faster.

4.1 Characterization of the Vectorized Loops

With these modifications, the inner pardo loops could be fully vectorized along the required rows or columns using scatter/gather operations. Note that while row access is by consecutive elements, column elements are accessed with a constant stride. In this section we describe the loops involved in each of the four phases and develop performance estimates for them. Of course, since the memory access patterns are data dependent, we can only give average case performance figures. However, these have shown to be fairly accurate predictors of performance.

The performance of a vector operation on the CRAY Y-MP may be characterized by the half-performance length ($n_{1/2}$) and the time per element to produce each result (t_e) [HJ88]. With these parameters the approximate time to execute a fully vectorized loop over n elements is

$$t(n) = t_e(n + n_{1/2}).$$

Table 3 summarizes the vector parameters for the loops of the four phases as described below. By using the indirect addressing scheme for the spinetree described in the previous section, the coding of each of these loops is straightforward.

1. The SPINETREE Loop

```
for (i = each element of row r) {
    spine[i] = bucket[label[i]];
    bucket[label[i]] = spine[i];
}
```

The loop shown above is issued for each row of elements. The compiler splits this (using loop fission) into a gather operation followed by a scatter.

2. The ROWSUM Loop

```
for (i = each element of column c) {
    rowsum[spine[i]] = rowsum[spine[i]] + value[i];
}
```

This loop is executed for each column. The compiler vectorizes this easily, accessing the elements of each column with a constant stride. However, since this loop involves 3 read operations and 1 write and there are only 2 read pipes on the Y-MP, it does not run at peak speed.

3. The SPINESUM Loop

```
for (i = each element of row r) {
    if (rowsum[i] != 0)
        spinesum[spine[i]] = rowsum[i] + spinesum[i];
}
```

This loop vectorizes but is problematic because of the technique the compiler currently uses. For each group of 64 elements, the compiler first determines which are FALSE. Each of these is given a dummy location to which to send a dummy value. In this way, elements whose ***rowsum*** is 0 do not update their parents. Because all dummy values are the same, when there are many FALSE sites the dummy location becomes a hot-spot for memory contention, possibly causing a performance loss. On the other hand, if all 64 elements are FALSE, none of the ***spine*** or ***spinesum*** values are even read and the loop jumps ahead to the next group of 64 elements. These two opposing effects lead to some strange results.

4. The PREFIXSUM Loop

```
for (i = each element of column c) {
    multi[i] = spinesum[spine[i]];
    spinesum[spine[i]] = spinesum[spine[i]] + value[i];
}
```

This loop is very similar to the one of the ROWSUM phase but involves one extra write. Because the CRAY Y-MP has only one write-pipe, this operation requires approximately the cost of an additional gather operation beyond the ROWSUM phase.

4.2 The Multireduce Operation

The multireduce operation provides only the reduction values for each label. By modifying the algorithm slightly we can obtain a multireduce operation that saves a significant amount of time over the entire multiprefix operation. The key insight is that after the SPINESUMS phase, the reduction values for each label may be calculated directly by summing the ***rowsum*** and ***spinesum*** value for each bucket.

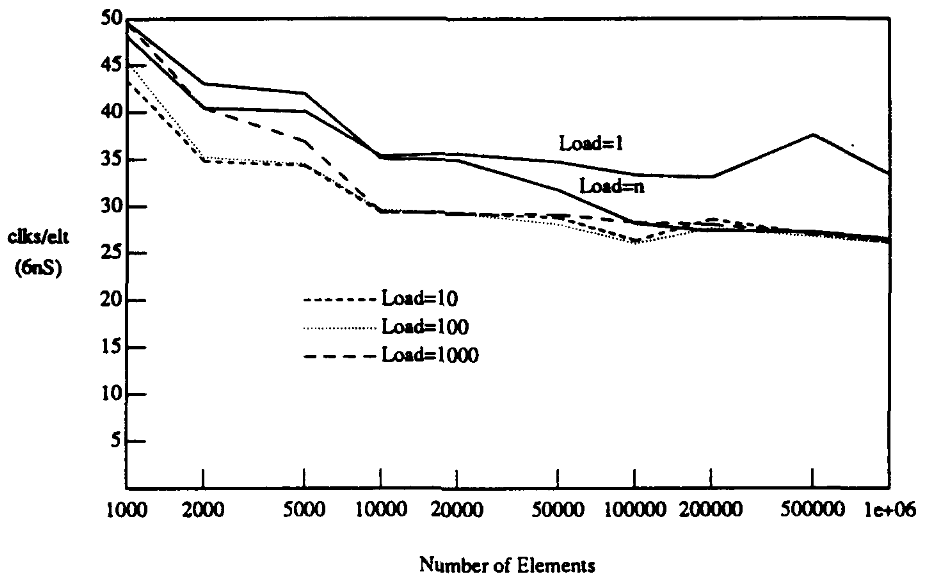

Figure 10: The time per element required for input sizes ranging from 1 thousand elements to 1 million. Each curve represents a different average bucket load. A load of n means that only one bucket was used and that all labels were the same. A load of 1 indicates that n labels were randomly distributed over n buckets. Other load factors represent more typical situations.

On the CRAY, this is a simple addition of two vectors and requires only slightly more than 1 clock tick per element. Compared to the PREFIXSUM phase, which requires almost 7 clock ticks per element, this is a substantial savings in time, for only a small modification to the algorithm.

4.3 Effects of Label Distribution

The performance of this algorithm is heavily dependent on the density and distribution of the integer labels. The "load" of a bucket is the number of elements in its class. While a heavy average load will degrade the performance of the SPINETREE phase, the same effect may cause the SPINESUM phase to run very quickly. In this section we will describe how these factors interact for different types of data for each of the four phases of the algorithm.

Using a standard pseudo-random number generator to provide us with labels, we timed the multiprefix operation over a wide range of input sizes and bucket load factors. Our results are summarized in Figure 10 with times expressed in 6nS clock ticks per element. Each curve represents a different bucket load factor. A load factor of n indicates that all elements had the same label, while a load factor of 1 means there were as many buckets as elements. However, because of the random number generator used, this does not indicate a one-to-one mapping from elements to buckets.

This data deserves detailed explanation. Three representative cases are explained in below. These are a heavy load (1 bucket), a moderate load, and a light load (n buckets).

Heavy Load: (All labels are the same.) The SPINETREE phase suffers because all scatter/gather operations are to the same memory location, requiring a total of 12 to 13 clock ticks per element. However, after this initially expensive operation, the other three phases run fairly quickly. The ROWSUMS and PREFIXSUMS phases perform as expected, but the SPINESUMS phase exhibits almost superlinear speedup.

Remember that the compiler breaks the rows into chunks equal to the vector length (VL), which is 64 on the CRAY Y-MP. Because only one element per row can be a spine element (since all are in the same class), only one group of 64 elements per row performs any real work, the other groups all exit early. For this reason, this loop runs in as little as 2 to 3 clocks ticks per element, offsetting the expensive spinetree creation phase.

Moderate Load: This is the region in which the algorithm is most predictable. The performance characterization numbers given earlier for the loops most accurately reflect this situation.

Light Load: Because our CRAY implementation initialized each of the buckets to 0 explicitly, this situation incurs the expense of the additional time required with a very large number of buckets. While the SPINETREE, ROWSUMS and PREFIXSUMS phases all run quite well for this case, it is again the SPINESUMS phase that behaves strangely. Because there are many classes of few elements each in this case, there are few spine elements on each row, but not so few as to gain the superlinear speedup effect. What occurs here is that each group of 64 elements has many FALSE sites which result in the writing of a dummy value to a dummy location. This one location receives values for ALL of the FALSE sites. Because of this memory contention, the SPINESUMS phase runs quite slowly, requiring 8 to 9 clock ticks per element.

What is most interesting about these observations is not the fact that performance varies with label distribution, but that the absolute performance of this algorithm shows little sensitivity to these variations. Even at the far extremes of heavy and light loading, the adverse affects to one phase are offset by the benefits to another. Over input sizes ranging many orders of magnitude, the time per element required varies no more than a few clocks. This fact should assure anyone using this algorithm that it will offer comparable performance for many different applications.

4.4 Choosing the row length

In the theoretical PRAM model of the preceding sections, the number of elements n was assumed to be a square. However, the length of the rows and columns may in fact be chosen separately so that their product is slightly greater than n. On the CRAY, the total time of the algorithm is more nearly a linear function of n provided the loops fully vectorize. Because the loops of the four phases are effectively vectrorized, each of the four main phases of the algorithm expend almost a constant amount of time per element.

The main body of the multiprefix algorithm executes in four loops alternatively over the rows and columns of the elements. Given a variable p as the chosen row length, the number of columns will be approximately n/p. The first phase, called SPINE, will then execute in time

$$t_1(n) = t_e^1(p + n_{1/2}^1)\frac{n}{p}$$

while the second phase, which cycles through the columns, will require

$$t_2(n) = t_e^2(\frac{n}{p} + n_{1/2}^2)n.$$

Ignoring the initialization phase, the time for the entire multiprefix algorithm is the sum over the four phases of the times required by each phase.

$$t_{\text{MP}} = t_e^1(p + n_{1/2}^1)\frac{n}{p} + t_e^2(\frac{n}{p} + n_{1/2}^2)n + t_e^3(p + n_{1/2}^3)\frac{n}{p} + t_e^4(\frac{n}{p} + n_{1/2}^4)n$$

We may find the value of p that minimizes this function by differentiating and setting the derivative equal to 0. This shows that the total time is minimized when

$$p = \sqrt{n}\sqrt{\frac{t_e^1 n_{1/2}^1 + t_e^3 n_{1/2}^3}{t_e^2 n_{1/2}^2 + t_e^4 n_{1/2}^4}}.$$

For the values of the loop parameters reported earlier, this gives

$$p = 0.749\sqrt{n}$$

indicating that a slightly shorter row length is preferred. (The reason this skewing is so slight is that the loop parameters are fairly evenly matched for the four phases.) However, the sensitivity of this formula to variations in p near the optimal value is very small. For example, using the average case loop performance figures from before, the percent difference between the total time with this optimal row length and a row length of $\sqrt{n}$ is less than 2% when $n = 1000$. For larger n, the total time is even less sensitive.

Because the row length p is the stride used for column access, a more important consideration is the choice of a value that minimizes memory bank conflicts. Our implementation chooses a value near the square root that is not a multiple of the number of memory banks nor of the bank cycle time (4 in the case of the CRAY Y-MP).

5 Applications Using Multiprefix on the CRAY Y-MP

In the introduction we showed the performance figures collected for two applications built using our implementation of the multiprefix operator on the CRAY Y-MP. Each uses multiprefix as their core step. This section describes the details of their implementation and provides further analysis.

5.1 Integer Sorting

The integer sorting problem requires sorting n integers keys whose values lie between 1 and m, for some known m. An algorithm for integer sorting using multiprefix was first described by Ranade in [RBJ88]. The algorithm computes a ***rank*** value for each key that gives its position in the final sorted order. Equivalently, the rank of each key indicates how many keys should precede it in sorted order. The entire algorithm is presented in Figure 11. Because the multiprefix operator guarantees that prefix sums are calculated in vector order, this sorting (ranking) algorithm is stable.

Using the integer keys, the first application of multiprefix-PLUS to a vector of 1's provides a count of the number of preceding equal keys for each integer. This step also leaves a count of the total number of each key in the buckets. For each bucket, its ***cumulative*** value gives the total number of lower-valued keys preceding it. This vector is calculated with another multiprefix-add

```
key:         int[n];
bucket:      int[m];
cumulative:  int[m];
rank:        int[n];

INITIALIZE:
    pardo (i = 1 to $m$)
        bucket[i] = 0;

INTEGER-SORT:
    MP(1, key, +, rank, bucket);
    MP(bucket, 1, total, cumulative);
    pardo (i = 1 to $n$)
        rank[i] = rank[i] + cumulative[key[i]] + 1;
```

Figure 11: A fast integer sorting algorithm using multiprefix.

operation with all keys being equal. By adding the prefix sums to these values, the final sorted ranking for each key is calculated.

Based on the previous complexity measures, this sorting algorithm has $S = O(\sqrt{n}+\sqrt{m})$ parallel step complexity, and performs $W = O(n + m)$ work. The serial counterpart to this algorithm is called "counting sort" and performs just as much work [Knu68, CLR89], so our algorithm is work efficient.

5.1.1 The NAS Integer Sorting Benchmark

In Table 1 we compared the time of our integer sorting procedure to two others recently reported. When timing the NAS integer sorting benchmark on the CRAY Y-MP we took advantage of the fact that each of these applications of multiprefix are simplified cases. In the first call to MP, the values summed are all 1. By using the fact that each *value[i]* was a constant, the compiler was able to generate more efficient code. This avoided a memory access in each of the ROWSUM and PREFIXSUM loops, allowing them to run faster.

The second multiprefix ADD is used to compute partial sums across the buckets, all of which have the same label. This operation is a simple prefix-sum, or recurrence computation of the form:

$$c_i = b_{i-1} + c_{i-1}.$$

For the benchmark timing, we resorted to the traditional "partition method" for solving this part of the problem [HJ88].

Even though both applications of multiprefix in this algorithm are simplified cases, its use is significant. Previous attempts to vectorize the first step of the bucket sorting algorithm have relied on sophisticated compiler technology to recognize this particlar loop. We made a simple change to a very general purpose algorithm and achieved excellent performance.

```
vector:  int[m];
rows:    int[n];
cols:    int[n];
vals:    double[n];
product: double[n];

PARALLEL-MATVECT:
    pardo (i = 1 to n)
        product[i] = vals[i] × vector[cols[i]];
    MR(product, rows, +, vector);
```

Figure 12: Sparse Matrix Vector multiplication using multiprefix.

5.2 Sparse Matrix Vector Multiplication

Multiplication of a dense vector by a sparse matrix is at the core of many numerical algorithms. This operation appears when solving systems of linear equations by iterative methods, and in finite element analysis.

The multireduce operator allows a straightforward sparse-matrix vector multiply algorithm as shown in Figure 12. A vector of length m is multiplied by an array with n non-zero elements. The elements are stored in three vectors that hold their values, and the row and column index of each. In the first step, all products are computed by multiplying each matrix element by the vector element matching its column index. Then, in the second step, all products with the same row index (key) are added together with the multireduce operator. (Because the partial sums are not needed, a full multiprefix is not used.)

Many elaborate storage schemes have been developed to allow the numerical portion of this algorithm to proceed at near peak speed on vector computers. The Compressed Sparse Row (CSR) storage format is most typically used and arranges the matrix into rows, with the column index of each element stored in a separate vector. This format is very simple and allows the matrix-vector multiply operation to vectorize completely over each row. However, for very sparse matrices, the row lengths can become quite short. Often they are much shorter than the vector half-length of the operation. In an attempt to write loops that better vectorize over more elements, other storage schemes have been developed.

The Jagged Diagonal (JD) format requires that the matrix is reordered so that the rows appear in decreasing order of population count. Here, the elements of the re-ordered matrix are collected into groups called "jagged-diagonals." The first jagged-diagonal consists of the first elements of each row; the second, of the second elements, etc. Because the rows are sorted, each of these groups is of successively decreasing length. The elements of the diagonals are stored in an array called JDA with their column positions in JDJ. The starting position of each jagged diagonal is given in an array the length of the dimension of the matrix called JDSTART, while the row index of each element is implicit in its position within each jagged-diagonal.

In this format, each jagged-diagonal begins with an element in row 1, and ends with an element in row k, where k is the length of the jagged-diagonal. Because each of the elements of a group are in different rows, each group may perform a vector update in parallel without the possibility of simultaneous access to the same vector element.

Many sparse matrices consist of just a few jagged-diagonals and the operation vectorizes completely over each jagged-diagonal. The disadvantage of the JD method is its large pre-processing time and the potential problems it has with non-uniform sparse matrices. For matrices with just a few long rows, many of the groups are very short and operations over them vectorize poorly.

5.2.1 Setup Time

The matrix-vector multiply operation can be divided into two parts. The first is the symbolic "setup" time, while the second is the numeric "evaluation" time. Often, when solving systems of linear equations, the same matrix multiplies a vector repeatedly. In this case, a high setup time can be amortized over many evaluations. It is precisely for this reason that the large setup time associated with the jagged-diagonal format is acceptible for some applications.

In the multiprefix approach, the setup time is precisely the time spent in the first phase of the multiprefix algorithm building the spinetree. We consider the CSR format approach the base case, and associate no setup time with it.

The times reported in the introduction in Table 2 are shown below in Table 4 broken down into setup, evaluation and total times. The order columns gives the number of rows of each matrix and the fraction of non-zero entries is expressed by its density, ρ. For very large sparse matrices the JD approach trades a large setup time for a quick evaluation. The multiprefix (MP) approach performs less of its total work during setup, while the CSR approach suffers from very short row lengths for extremely sparse matrices. Because of the speed of the evaluation phase of the JD approach, its use would be preferable in an application that requires repeated multiplication of the same matrix, while the MP approach would be better suited to cases where only one multiplication is performed. However, the JD approach has problems with non-uniform matrices.

A different series of trials was timed using matrices from electrical circuit simulation problems distributed with the SPARSE sparse matrix package [KSV]. These matrices are very sparse, with an average of only 7 or 8 elements per row, but have a few very long rows. These rows represent power and ground and are almost completely populated. The results are presented in Table 5.

In these few trials the MP approach clearly In these cases the MP approach clearly outperforms both the CSR and JD weakness of the JD format, and it has been suggested that such long rows should be handled as a special case [AY89]. In general, the performance of the multiprefix approach is more consistent over matrices of varying structure.

6 Conclusions

In this paper we presented a parallel algorithm that implements the multiprefix operation. Our approach introduces some novel techniques. We used the power of the arbitrary concurrent write as an arbitrarion scheme to build the virtual combining network represented by our SPINETREE data structure. Then, having designed a general purpose synchronous parallel algorithm, we ported it to the CRAY Y-MP by simulating each parallel step with a single vector operation. Since our parallel algorithm is work efficient, we are assured that our fully vectorized simulation of the algorithm runs in linear time.

The multiprefix operation has been proposed as a general purpose parallel primitive. We showed how it can be used to build two seemingly unrelated algorithms: integer sorting and sparse-matrix vector multiplication. By using the multiprefix operator, the algorithms are written at a very high level, avoiding the detailed loop organizations often developed for vectorized algorithms. Using this

Table 4: A breakdown of the setup and evaluation times of three approaches to sparse matrix vector multiplication for matrices of varying size and density. For each category, the best times are highlighted. The CSR approach does no preprocessing, while the JD approach trades a large preprocessing time for a very quick evaluation time. The TOTAL time represents the time required to perform one setup and evaluation. When performing only one matrix vector multiply, the multiprefix approach excels for very sparse matrices.

Sparse Matrix Vector Multiplication (times in mS)										
Order	ρ	Setup			Evaulation			Total		
		CSR	JD	MP	CSR	JD	MP	CSR	JD	MP
15000	0.001		24.26	**5.87**	30.29	**3.83**	21.56	30.29	28.09	**27.43**
10000	0.001		14.58	**2.64**	19.52	**1.73**	9.79	19.52	16.31	**12.43**
5000	0.001		6.54	**0.81**	9.48	**0.45**	2.64	9.48	6.99	**3.45**
2000	0.005		2.90	**0.65**	3.90	**0.33**	2.12	3.90	3.23	**2.77**
1000	0.010		1.47	**0.36**	1.95	**0.19**	1.14	1.95	1.66	**1.50**
100	0.400		0.32	**0.20**	0.27	**0.10**	0.56	**0.27**	0.42	0.76
50	1.000		0.19	**0.24**	0.14	**0.13**	0.29	**0.14**	0.32	0.53

Table 5: A comparison of the setup and evaluation times for some matrices representing electrical circuits. For these matrices with a few very full rows, the JD approach suffers a severe performance loss.

Sparse Matrix Vector Multiplication (times in mS)										
Title:	Order	ρ	Setup			Evaulation			Total	
			CSR	JD	MP	CSR	JD	MP	CSR	JD	MP
ADVICE2806	2806	.0030		6.08	**1.85**	7.99	2.41	**2.28**	7.99	8.49	**4.13**
ADVICE3776	3776	.0019		8.13	**2.10**	7.19	3.21	**2.72**	7.19	11.34	**4.82**

approach we developed algorithms competetive in performace with more traditional FORTRAN-based approaches.

This work provides further evidence in support of higher-level parallel primitives for portable parallel programming. By structuring algorithms at a more abstract level we relieve the programmer from writing machine-dependent code or adding compiler directives that show where the parallelism is. In the long run, as parallel computer architectures evolve, only the implementations of the parallel primitives will be refined, allowing user application code to be reused.

7 Acknowledgments

I would like to thank my advisor, Randy Bryant, for suggesting that I investigate this problem and for his advice as this work evolved.

References

[AY89] E. Anderson and Y.Saad. Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing*, 1:73–96, 1989.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinki, R. S. Schreiber, H. D Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks — summary and preliminary results. In *Proceedings Supercomputing '91*, 1991.

[BBCS91] D. Bailey, D. Browning, R. Carter, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[Ble90] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, 1990.

[CBZ90] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, November 1990.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1989.

[Coh90] Evan Reid Cohn. Implementing the multiprefix operation efficiently. *Journal of Parallel and Distributed Computing*, 10:29–34, 1990.

[GLR81] A. Gottlieb, B.D Lubachevsky, and L. Rudolph. Coordinating large numbers of processors. In *Proc. 1981 International Conference on Parallel Processing*, 1981.

[Hil85] W. Daniel Hillis. *The Connection Machine.* The MIT Press series in artificial intelligence. MIT Press, Cambridge, Mass., 1985.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: architecture, programming and algorithms.* IOP Publishing Ltd., 1988.

[Kan90] Yasusi Kanada. A vectorization technique of hashing and its application to several sorting algorithms. In *PARBASE-90, International Conference on Databases, Parallel Architectures, and Their Applications*, 1990.

[Knu68] Donald Knuth. *The Art of Computer Programming; Volume 3: Sorting and Searching.* Computer Science and Information Processing. Addison-Wesley, 1968.

[KSV] Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli. Sparse 1.3, a sparse linear equation solver. Available from EECS Industrial Liaison Program, University of California, Berkeley, CA 94720.

[PMM92] Douglas M. Pase, Tom MacDonald, and Andrew Meltzer. MPP Fortran programming model. Technical report, Cray Research, Inc., January 1992.

[RBJ88] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnsson. The Fluent abstract machine. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI, Proceedings of the Fifth MIT Conference*, pages 71–93, 1988.

[Saa89] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM J. on Stat. Scient. Computing*, 10:1200–1232, 1989.

[SH86] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine Lisp: Fine-grained parallel symbolic processing. Technical Report 86.16, Thinking Machines Corporation, May 1986.