

Scan Primitives for Vector Computers*

Siddhartha Chatterjee

Guy E. Blelloch

Marco Zagha

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

This paper describes an optimized implementation of a set of *scan* (also called all-prefix-sums) primitives on a single processor of a CRAY Y-MP, and demonstrates that their use leads to greatly improved performance for several applications that cannot be vectorized with existing compiler technology. The algorithm used to implement the scans is based on an algorithm for parallel computers and is applicable with minor modifications to any register-based vector computer. On the CRAY Y-MP, the asymptotic running time of the *plus-scan* is about 2.25 times that of a vector add, and is within 20% of optimal. An important aspect of our implementation is that a set of *segmented* versions of these scans are only marginally more expensive than the unsegmented versions. These segmented versions can be used to execute a scan on multiple data sets without having to pay the vector startup cost ($n_{1/2}$) for each set.

The paper describes a radix sorting routine based on the scans that is 13 times faster than a Fortran version and within 20% of a highly optimized library sort routine, three operations on trees that are between 10 and 20 times faster than the corresponding C versions, and a connectionist learning algorithm that is 10 times faster than the corresponding C version for sparse and irregular networks.

1 Introduction

Vector supercomputers have been used to supply the high computing power needed for many applications. However, the performance obtained from these machines critically depends on the ability to produce code that vectorizes well. Two distinct approaches have been taken to meet this goal—vectorization of “dusty decks” [18], and language support for vector intrinsics, as seen in the proposed Fortran 8x standard [1]. In both cases, the focus of the work has been in speeding up “scientific” computations, characterized by regular and static data structures and predominantly regular access patterns within these data structures. These alternatives are not very effective for problems that

*This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Routine	t_r (clks/elt)	
	Scan version	Scalar version
plus scan (int)	2.5	7.5 (*)
segmented plus scan (int)	3.7	74.8 (*)
parallel radix sort (64 bits)	896.8	11730.0 (*)
branch_sums	12.2	206.5 (†)
root_sums	9.8	208.6 (†)
delete_vertices	19.2	276.2 (†)

Table 1: Incremental processing times per element for primitives and applications discussed in this paper, for both scan and scalar versions. All numbers are for a single processor of a CRAY Y-MP. 1 clock tick = 6 ns. Items marked with (*) were written in Fortran and those marked with (†) were written in C.

create and manipulate more irregular and dynamically varying data structures such as trees and graphs.

Elsewhere, scan (prefix sum) operations have been shown to be extremely powerful primitives in designing parallel algorithms for manipulating such irregular and dynamically changing data structures [3]. This paper shows how the scan operations can also have great benefit for such algorithms on pipelined vector machines. It describes an optimized implementation of the scan primitives on the CRAY Y-MP¹, and gives performance numbers for several applications based on the primitives (see Table 1). The approach in the design of these algorithms is similar to that of the Basic Linear Algebra Subprograms (BLAS) developed in the context of linear algebra computations [14] in that the algorithms are based on a set of primitives whose implementations are optimized rather than having a compiler try to vectorize existing code.

The remainder of this paper is organized as follows. Section 2 introduces the scan primitives and reviews previous work on vectorizing scans. Section 3 discusses our implementation in detail and presents performance numbers. Section 4 presents other primitives used in this paper, and three applications using these primitives. Finally, future work and conclusions are given in Section 5.²

¹CRAY Y-MP and CFT77 are trademarks of Cray Research, Inc.

²All Fortran code discussed in this paper was compiled with CFT77 Version 3.1. All C code was compiled with Cray PCC Version XMP/YMP 4.1.8.

2 Scan primitives

A scan primitive takes a binary associative operator \oplus with left identity I_\oplus , and a sequence $[a_0, \dots, a_{n-1}]$ of n elements, and returns the sequence

$$[x_0, \dots, x_{n-1}] = [I_\oplus, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus \dots \oplus a_{n-2})].$$

In this paper, we will only use $+$ and **maximum** as operators for the scan primitives, as these operators are adequate for all the applications discussed here. We henceforth refer to these scan primitives as **+-scan** and **max-scan** respectively. For example:

$$\begin{aligned} A &= [5 & 1 & 3 & 4 & 9 & 2] \\ \text{+-scan}(A) &= [0 & 5 & 6 & 9 & 13 & 22] \\ \text{max-scan}(A) &= [-\infty & 5 & 5 & 5 & 5 & 9] \end{aligned}$$

The scan primitives come from APL [10]. Livermore loop 11 is a scan with addition as the binary associative operator.

We call the scans as defined above *unsegmented* scans. Many applications execute the scan operations on multiple data sets. Rather than paying a large startup cost for each scan, it is desirable to put the sets in adjacent segments of memory and use flags to separate the segments. For example:

$$\begin{aligned} A &= [5 & 1 & 3 & 4 & 3 & 9 & 2] \\ \text{head-flags} &= [1 & 0 & 0 & 1 & 0 & 1 & 0] \\ \text{seg-+-scan}(A) &= [0 & 5 & 6 & 0 & 4 & 0 & 9] \end{aligned}$$

The segmented versions of the scan execute a scan independently on each segment, but only pay the startup cost ($n/2$) once. Section 3.3 discusses the tradeoff, and Section 4.3 gives an example of how they are used.

The uses of both unsegmented and segmented scans are extensive. A list of some of them is given below.

1. To add arbitrary precision numbers. These are numbers that cannot be represented in a single machine word [17].
2. To evaluate polynomials [19].
3. To solve recurrences such as $x_i = a_i x_{i-1} + b_i x_{i-2}$ or $x_i = a_i + b_i/x_{i-1}$ (Livermore loops 6, 11, 19 and 23) [12].
4. To solve tridiagonal linear systems (Livermore Loop 5) [20].
5. To pack an array so that marked elements are deleted (the **PACK** intrinsic in Fortran 8x).
6. To find the location of the first minimum in an array (Livermore loop 24, and the **MINLOC** intrinsic in Fortran 8x).
7. To implement a radix sorting algorithm (see Section 4.1).
8. To implement a quicksort algorithm [3].
9. To perform lexical analysis [7].
10. To implement operations on trees, such as finding the depth of every vertex in a tree (see Section 4.2).

$S = [[s_{00} \ s_{01} \ s_{02}] \ [] \ [s_{20} \ s_{21}] \ [s_{30} \ s_{31}]]$							
vector elements	=	$[s_{00} \ s_{01} \ s_{02} \ s_{20} \ s_{21} \ s_{30} \ s_{31}]$					
head-flags	=	$[1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0]$					
lengths	=	$[3 \ 0 \ 2 \ 2]$					
head-pointers	=	$[0 \ 3 \ 3 \ 5]$					

Figure 1: A segmented vector and three ways of describing the segmentation. The *head-flags* marks the beginning of each segment, the *lengths* specifies the length of each segment, and the *head-pointers* points to the beginning of each segment.

11. To label components in two-dimensional images [16].
12. To draw a line on a two-dimensional grid given its endpoints [4].
13. To solve a variety of graph problems [2].
14. To solve many problems in computational geometry [4].
15. To efficiently implement languages that support nested parallel structures [5], such as CM-LISP [23], on flat parallel machines.

Comparison with previous work

There has been significant research on parallel and vector algorithms for unsegmented scans. A parallel implementation of scans on a perfect shuffle network was suggested by Stone [19] for polynomial evaluation. The problem with this algorithm is that it required $O(\log n)$ time with $O(n)$ processors. Thus it required $O(n \log n)$ arithmetic operations as opposed to $O(n)$ for the serial algorithm. Stone later showed an optimal version of the algorithm in the context of solving tridiagonal equations [20]. This algorithm can be implemented in $O(\log n)$ time with $O(n/\log n)$ processors. When implemented on the CRAY Y-MP, this algorithm results in running times that are longer than those for the algorithm presented in this paper, since it requires intermediate results to be written out to main memory and then read back in again. Other approaches to implementing scans on a vector machine are based on cyclic reduction [8] and unrolling scalar loops [21], which again have larger constant factors than the implementation presented in the paper.

The frequent use of linear recurrences in scientific code has motivated research in finding efficient and vectorizable methods for scans. For instance, Livermore loops 5, 6, 11, 19 and 23 involve recurrences that can be expressed in terms of scans. State-of-the-art vectorizing compilers can recognize such linear recurrences and use one of the above algorithms to optimize such loops [15]. However, segmented scans cannot be handled by such techniques. This is the reason that the performance of the unsegmented scalar plus scan in Table 1 is significantly better than that of the segmented plus scan.

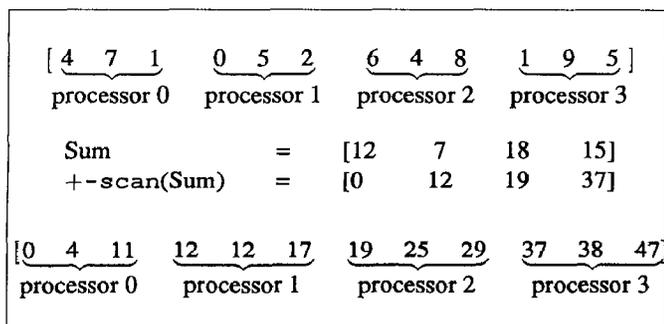


Figure 2: Executing a scan when there are more elements than processors.

Special hardware support for summation and iteration type operators has been added on some vector machines, the Hitachi S-820 being an example [22]. This machine includes a feedback connection from the output of the ALU to the inputs to support such operations. However, the speed of these instructions is much slower than those of other vector instructions, and efforts to vectorize scans based on such primitives [21] have only been able to realize a very small fraction of the peak performance. The techniques suggested there cannot handle segmented scans either. Our work demonstrates that such specialized hardware is not essential, and that effective use can be made of existing hardware to vectorize scan operations.

Loop interchange [18] is a technique used by vectorizing compilers to vectorize inner loops by interchanging the order of enclosing loops. It appears that the functionality provided by segmented scans can be achieved through loop interchange techniques. This is true for operations on arrays, where the segments (corresponding to rows or columns of the array) are of equal length. However, such techniques would not be applicable to more irregular data structures such as graphs or sparse matrices, where the segments are not necessarily of the same length. Performance there would be dominated by the length of the longest segment. Further, the performance of the loop interchange technique depends on the number of segments and is bad for a small number of segments, whereas the performance of the segmented scan will be shown to be independent of this number.

3 Implementing Scan Primitives

In this section, we present vectorizable algorithms for the unsegmented and segmented scan primitives. We only consider the +-scan, as the algorithms can be easily modified for max-scan by changing the addition operation to a maximum.

We first describe a parallel algorithm for plus-scanning a vector of length n on p processors. It has three phases:

1. Processor i sums elements i_s through $(i + 1)s - 1$, where $s = n/p$. (For simplicity assume that n is a multiple of p .) Call this result Sum[i].
2. The vector Sum is now scanned using a tree-summing [13] or similar algorithm.

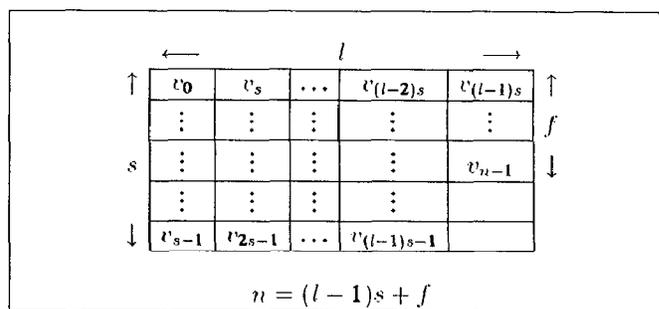


Figure 3: Rectangular organization of a vector of length n for scan operations. The vector is fully characterized by specifying the shape factors l , s and f . Each cell represents one element of the vector. On the CRAY Y-MP, l ($0 < l \leq 64$) is the length of the vector registers used in the computation. s is the stride used to load the vector register from memory, and also the number of iterations of each phase of the algorithm. f is the number of iterations that are executed with a vector register length of l . The remaining $s - f$ iterations are executed with the length set to $l - 1$.

3. Processor i sums its portion of the vector, starting with Sum[i] as the initial value, and accumulating partial sums. The partial sums are written out to the corresponding locations.

Figure 2 shows an example (the details of the tree-summing step are omitted).

To vectorize the parallel algorithm, consider each element of a vector register as a processor. Then, in the above scheme, element i of the vector register would need to sum elements i_s through $(i + 1)s - 1$ of the vector being scanned. This operation can be vectorized by traversing the vector with stride s , and using elementwise vector instructions to sum the sections. It is useful to consider the vector (of n elements) as being organized in a rectangular fashion, characterized by three parameters l , s and f , as shown in Fig. 3. These three parameters characterize the vector completely and are called its *shape factors*.

Determining the shape factors

From Fig. 3, $n = (l - 1)s + f$. This equation is underspecified for the purposes of determining the shape factors given n . We use the following additional constraints for choosing the parameters:

- $0 < f \leq s$.
- s is chosen to minimize bank conflicts; it is not allowed to be a multiple of BANKS, a constant that is determined by the number of memory banks and the bank cycle time (4 in the case of the CRAY Y-MP).
- l is kept as large as possible, and is upper-bounded by the length of the vector registers (64 in the case of the CRAY Y-MP).

The reason for organizing the data thus is that all memory accesses are now constant-stride, and bank conflicts are avoided.

We refer to this method of traversal of the vector as *scan order*, and the usual unit-stride traversal as *index order*. Thus, for a vector with $l = 4$ and $s = f = 3$, a scan order traversal would read the elements in the order ((0, 3, 6, 9), (1, 4, 7, 10), (2, 5, 8, 11)), whereas an index order traversal would read them in the order ((0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11)).

3.1 Algorithm for unsegmented plus-scan

The unsegmented scan algorithm mirrors the parallel algorithm and has three phases. In the following discussion we will assume $f = s$ for purposes of clarity. If $f < s$, the algorithm needs to be modified to decrement the contents of the Vector Length register by 1 for the last $(s - f)$ iterations. This algorithm can be used on any register-based vector machine. In our implementation on the CRAY Y-MP, the loops shown below are unrolled and the registers renamed to avoid locking vector registers. This unrolling speeds up phase 3 by a factor of two.

1. (Phase 1) Elements are summed columnwise, producing l sums, thus:

$$\begin{aligned} V1_0 &= v_0 + \dots + v_{s-1} \\ V1_1 &= v_s + \dots + v_{2s-1} \\ &\vdots \\ V1_{l-2} &= v_{(l-2)s} + \dots + v_{(l-1)s-1} \\ V1_{l-1} &= v_{(l-1)s} + \dots + v_{n-1} \end{aligned}$$

The sums $V1_i$ are stored in the elements of a vector register, which is initialized to zero. The entire computation is vectorizable, as the following C-like pseudo-code shows. In this code, V1 and V2 are vector registers, and the vector length register is set to l . V1 is used to compute and hold the columnwise sums, and V2 is used to load sections of the source vector.

```
V1 = 0;
k = (l-1)*s;
for (i = 0; i < s; i++) {
    /* vector fetch with stride s */
    V2 = <v[i], v[s+i], ..., v[k+i]>;
    V1 = V1 + V2; /* vector add */
}
```

2. (Phase 2) A serial scan is performed on the elements of vector register V1. After this computation, the new values of the elements are as follows:

$$\begin{aligned} V1_0 &= 0 \\ V1_1 &= \sum_{i=0}^{s-1} v_i \\ &\vdots \\ V1_{l-2} &= \sum_{i=0}^{(l-2)s-1} v_i \\ V1_{l-1} &= \sum_{i=0}^{(l-1)s-1} v_i \end{aligned}$$

Src	=	[2	4	1	1	0	1	-3	2	0	6	1	5]
Vector organization	=	2	1	-3	6								
		4	0	2	1								
		1	1	0	5								
V1 (in phase 1)	=	0	0	0	0	initially							
		2	1	-3	6	after iter 1							
		6	1	-1	7	after iter 2							
		7	2	-1	12	after iter 3							
V1 (after phase 2)	=	0	7	9	8								
Dst (in phase 3)	=	0	7	9	8	in iter 1							
		2	8	6	14	in iter 2							
		6	8	8	15	in iter 3							
Dst	=	[0	2	6	7	8	8	9	6	8	8	14	15]

Figure 4: Stages in the execution of the unsegmented scan operation. The diagram shows the organization of the source vector (Src), the intermediate states of V1 during phase 1 of the scan, the values in V1 after phase 2, and the values written out to the destination vector (Dst) during phase 3.

This computation is not vectorized, but is of fixed cost since it only operates on l elements ($l \leq 64$ on the CRAY Y-MP). Moreover, it executes out of the vector register and does not involve any memory transfers.

3. (Phase 3) The elements are again summed columnwise, but using the current values in V1 as an initial offset, as follows:

```
k = (l-1)*s;
for (i = 0; i < s; i++) {
    /* vector fetch with stride s */
    V2 = <v[i], v[s+i], ..., v[k+i]>;
    /* vector store with stride s */
    <d[i], d[s+i], ..., d[k+i]> = V1;
    V1 = V1 + V2; /* vector add */
}
```

This computation is also vectorizable. The values stored into d are the values resulting from the plus-scan. The vectors d and v can be identical; this does not affect the correctness of the algorithm.

Figure 4 shows the various stages of the above example. The vector v on which the scan is performed is of length 12, which, for purposes of illustration, is organized with $l = 4$ and $s = f = 3$. It should be noted that the first two phases of the above algorithm can be used to implement a standard reduction (or summation) operation.

V_0	V_s	V_{2s}	...	$V_{(l-1)s}$	0^{64-l}
V_1	V_{s+1}	V_{2s+1}	...	$V_{(l-1)s+1}$	0^{64-l}
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
V_{f-1}	$V_{s+(f-1)}$	$V_{(l-1)s+(f-1)}$	0^{64-l}
V_f	V_{s+f}	...	$V_{(l-2)s+f}$	0	0^{64-l}
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
V_{s-1}	V_{2s-1}	...	$V_{(l-1)s-1}$	0	0^{64-l}

Figure 5: Compressed transposed representation of segment flags. Each cell represents a bit, except the last column, which represents the padding consisting of 0-bits. Each row corresponds to one 64-bit word, so that the compressed representation requires s words of storage.

3.2 Algorithm for segmented plus-scan

For this primitive, segments are identified by a bit-vector of flags, that is set to 1 at the positions where segments start, and 0 elsewhere. On the CRAY Y-MP, we store the flags in a packed transposed representation that is similar to the rectangular representation of vectors, except that each element is 1 bit long. This means that the flags can be stored in s words, as shown in Fig. 5. This reduces the storage required for the flags, and doubles the speed of the code by making effective use of the Vector Mask register. This algorithm, while specific to the CRAY Y-MP, can be easily modified to run on other vector machines.

The implementation of the segmented plus-scan is very similar to that of the unsegmented version, with one change to handle segments. This involves resetting the running sum to zero whenever the start of a new segment is reached. The flags are assumed to be in the compressed form as shown in Figure 5. The three phases of the algorithm are as follows. In the description below, V1 and V2 are vector registers, S1 is a scalar register that is used to store flag information, F is the array containing the compressed flags, and VM is the vector mask register. Once again, loop unrolling and register renaming are required to avoid locking vector registers.

1. (Phase 1) We present only the pseudo-code for this phase. The merge operation is directly supported by the CRAY Y-MP. The two scalar instructions can run overlapped with the vector instructions, which themselves run in a chained fashion.

```

V1 = 0;
S1 = 0;
k = (l-1)*s;
for (i = 0; i < s; i++) {
    /* vector fetch with stride s */
    V2 = <v[i], v[s+i], ..., v[k+i]>;
    /* flag fetch (scalar) */
    VM = F[i];
    /* reset V1 if F is 1 (vector) */
    V1 = V1 merge 0 with VM;
    V1 = V1 + V2; /* vector add */
}

```

```

Src = [2 4 1 5 8 1 3 2 3 6 0 5]
Seg = [1 0 0 0 0 1 1 0 1 0 0 0]

```

```

Vector organization = 2 5 3 6
                    4 8 2 0
                    1 1 3 5

```

```

Flags organization = 1 0 1 0
                   0 0 0 0
                   0 1 1 0

```

```

V1 (in phase 1) = 0 0 0 0    initially
                2 5 3 6    after iter 1
                6 13 5 6   after iter 2
                7 1 3 11   after iter 3

```

```

S1 (in phase 1) = 0 0 0 0    initially
                1 0 1 0    after iter 1
                1 0 1 0    after iter 2
                1 1 1 0    after iter 3

```

```

V1 (after phase 2) = 0 7 1 3

```

```

Dst (in phase 3) = 0 7 0 3    in iter 1
                 2 12 3 9    in iter 2
                 6 0 0 9    in iter 3

```

```

Dst = [0 2 6 7 12 0 0 3 0 3 9 9]

```

Figure 6: Stages in the execution of the segmented scan operation. The diagram shows the organizations of the source vector (Src) and the flags (Seg), the intermediate states of V1 and S1 during phase 1 of the scan, the values in V1 after phase 2, and the values written out to the destination vector (Dst) during phase 3. The flags are bits, and all the other vectors are integers. The padding on words of the compressed flags is omitted for clarity.

```

/* OR flag bits (scalar) */
S1 = S1 | VM;
}

```

2. (Phase 2) The pseudo-code for this phase is as follows:

```

s = 0;
for (i = 0; i < l; i++) {
    /* V1[i] is the i'th elt
       of register V1 */
    t = V1[i];
    V1[i] = s;
    /* S1<i> is the i'th bit
       of S1 */
    if (S1<i> != 0) s = 0;
    s = s + t;
}

```

This loop runs in scalar mode, but the cost is small and fixed, and it involves no memory transfers.

3. **(Phase 3)** This phase is similar to phase 1 except that the results are written out and the flags need not be ORed together.

```

k = (l-1)*s;
for (i = 0; i < s; i++) {
    /* vector fetch with stride s */
    V2 = <v[i], v[s+i], ..., v[k+i]>;
    /* flag fetch (scalar) */
    VM = F[i];
    /* reset V1 if F is 1 (vector) */
    V1 = V1 merge 0 with VM;
    /* vector put with stride s */
    <d[i], d[s+i], ..., d[k+i]> = V1;
    V1 = V1 + V2; /* vector add */
}

```

The packed representation for flags allows successive words to be loaded directly into the Vector Mask register, and a vector merge can be used to reset the values of V. In practice, we cache 64-word sections of the flags in a vector register to cut down on memory access costs.

Figure 6 shows the various stages of the above example. The vector v on which the scan is performed is of length 12, which, for purposes of illustration, is organized with $l = 4$ and $s = f = 3$. The vector has four segments, of lengths 5, 1, 2, and 4 respectively.

Algorithm for segmented copy-scan

Other useful segmented scan operations include max- and copy-scans. A segmented copy-scan copies values placed at the starting positions of segments across the segment, and can be implemented in a manner similar to the segmented plus-scan³. The copy-scan copies either the running value or the new value based on the flag at that point, in the same way that the plus-scan either adds the value to the running sum or clears the sum. The details are similar to the segmented plus-scan and are therefore not repeated here.

3.3 Results

Table 2 presents timing results for our implementations of the scan primitives. We characterize the performance of each operation with two numbers, t_c ($= 1/r_{\infty}$) and $n_{1/2}$ [9], that measure, respectively, the incremental cost of processing one element (in clock cycles), and the vector half-performance length. Our efforts have been mostly concerned with reducing t_c as this is (asymptotically) the dominant factor in the running time.

The primitives are implemented in Cray Assembler Language (CAL) [6] as a set of routines callable from the C programming language. The loops are unrolled twice to avoid register

³This is not a scan as defined in Section 2, but is so called because it can be implemented in a similar manner.

Operation	t_c (clocks/element)	$n_{1/2}$ (elements)
plus_scan_i	2.49	855
max_scan_i	4.75	602
seg_copy_scan_i	3.88	893
seg_plus_scan_i	3.65	827

Table 2: t_c ($= 1/r_{\infty}$) and $n_{1/2}$ values for various scan primitives. The values are for a single processor of a CRAY Y-MP.

reservation effects. Timing measurements were taken with the `rtclock()` system call, and multiple measurements were taken to smooth out the effects of the presence of other users on the system. A least-squares technique was used to fit a straight line to the points and calculate t_c and $n_{1/2}$.

Theoretically, t_c for both the unsegmented and the segmented plus-scan on the CRAY Y-MP for our algorithm is 2 cycles/element, assuming perfect chaining. Our implementation of the unsegmented scan achieves a rate of 2.5 cycles/element, which is 80% of the peak throughput. The segmented scan implementation achieves 3.65 cycles/element, 55% of the peak rate.

Segmented vs. unsegmented scans

It is clear that the effect of a segmented scan can be simulated by performing an unsegmented scan over each of its segments. We show that this is generally undesirable. Consider a vector of length n containing m segments. The execution time t of a vector operation is given by the following equation [9]:

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

Using superscripts s and u to indicate, respectively, the segmented and unsegmented versions of the scan operation, the times to execute the segmented scan operation by the two methods above are given by:

$$\begin{aligned}
 t^s &= t_c^s(n + n_{1/2}^s) \\
 t^u &= t_c^u(n + m \cdot n_{1/2}^u)
 \end{aligned}$$

The set of unsegmented scans runs faster when:

$$m < \frac{t_c^s \cdot n_{1/2}^s}{t_c^u \cdot n_{1/2}^u} + \frac{t_c^s - t_c^u}{t_c^u \cdot n_{1/2}^u} n,$$

which, on substituting values from Table 2, gives

$$m < 1.42 + 0.0005n.$$

This indicates that the unsegmented algorithm should only be used to implement a segmented scan if there are a small number of relatively long segments.

4 Applications

In this section we discuss three applications using these primitives: sorting, tree operations, and a connectionist learning algorithm called back-propagation. These applications were chosen

Operation	Eqv. Fortran kernels
Elementwise	
p_add_i(C, A, B, L)	C(I) = A(I) + B(I)
p_sub_i(C, A, B, L)	C(I) = A(I) - B(I)
p_neg_i(B, A, L)	B(I) = -A(I)
p_max_i(C, A, B, L)	IF (A(I) .GT. B(I)) THEN C(I) = A(I) ELSE C(I) = B(I)
p_inv_i(B, A, L)	IF (A(I) .EQ. 0) THEN B(I) = 1 ELSE B(I) = 0
ext_bit(C, A, N, L)	C(I) = _gbit(A(I), 64-N)
p_sel_i(C, A, B, F, L)	IF (F(I) .EQ. 0) THEN C(I) = A(I) ELSE C(I) = B(I)
p_distr_i(A, N, L)	A(I) = N
Permutation-based	
perm_i(C, A, B, L)	C(B(I)) = A(I)
bperm_i(C, A, B, L)	C(I) = A(B(I))
sel_perm_i(C, A, B, F, L)	IF (F(I) .EQ. 1) C(B(I)) = A(I)

Table 3: Other vector primitives used. The kernel loops over the L elements of the vectors. `_gbit` is an intrinsic function to extract a bit from a word. Some of the kernels cannot be vectorized without a change in the algorithm.

to illustrate the use of the scan primitives in problems that are not regular.

We first introduce a set of primitives in addition to scans that we have implemented as building blocks for applications. These primitives are shown in Table 3 along with their semantics. The elementwise operations correspond directly to elementwise vector instructions found on the Cray. The permutation-based operations are implemented using the gather and scatter instructions. There is obviously a wide variety of operations to choose from in selecting the particular set. The operations we considered primitive were based on considerations of generality and ease of implementation. They were implemented in CAL as the C compiler did not do a good job of vectorizing them.

Table 4 shows the performance of the other primitives, in terms of t_e and $n_{1/2}$. The performance of the permutation-based operations are data-dependent because of the possibility of memory bank conflicts. We timed identity, reverse and unstructured permutations.

4.1 Sorting

As our first example of the use of the scan primitives, consider a simple radix sorting algorithm. The algorithm is a parallel version of the standard serial radix sort [11].

The algorithm loops over the bits of the keys, starting at the lowest bit, executing a `split` operation on each iteration. The `split` operation packs the keys with a 0 in the corresponding

Operation	t_e (clks/elt)	$n_{1/2}$ (elts)
p_add_i	1.14	483
p_sub_i	1.16	767
p_neg_i	1.11	710
p_max_i	2.12	398
p_inv_i	1.11	700
ext_bit	1.11	709
p_sel_i	2.13	483
p_distr_i	1.09	668
perm_i		
(identity)	1.24	663
(reverse)	1.60	499
(unstructured)	1.82	419
bperm_i		
(identity)	1.56	557
(unstructured)	1.87	446
sel_perm_i		
(identity)	3.39	332

Table 4: $t_e (= 1/r_\infty)$ and $n_{1/2}$ values for other vector primitives. The numbers are for a single processor of a Cray Y-MP.

bit to the bottom of a vector, and packs the keys with a 1 in the bit to the top of the same vector. It maintains the order within both groups. The sort works because each `split` operation sorts the keys with respect to the current bit (0 down, 1 up) and maintains the sorted order of all the lower bits—remember that we iterate upwards from the least significant bit. Figure 7 shows an example of the sort along with code to implement it.

We now consider how the `split` operation can be implemented using the scan primitives (see Figure 8). The basic idea is to determine a new index for each element and permute the elements to these new indices. This can be done with a pair of scans as follows. We plus-scan the inverse of the `Flags` into a vector `I_down`. This gives the correct offset for all the elements with a 0 flag. We now generate the offsets for the elements with a 1 and put them in `I_up`, by executing a plus-scan on the flags (we must first add to the initial element the number of 0 elements). We then select the indices from the two vectors `I_up` and `I_down` based on the original flags vector. Figure 8 shows our code that implements the `split` operation. Figure 9 shows a Fortran implementation of the `split` operation. The available Fortran compiler did not vectorize this code.

The performance of the parallel radix sort on 32- and 64-bit inputs is shown in Table 5. This is within 20% of a highly optimized library sort routine and 13 times faster than a sort written in Fortran.

On the Cray Y-MP, the `split` operation can be implemented with the `compress-index` vector instruction with about the same performance (t_e) as our implementation. It is interesting that we can get almost the same performance without ever using this instruction which is designed specifically for `split` type operations.

```

for (bit = 0; bit < num_bits; bit++) {
  ext_bit(Flags, V, bit, len);
  split(V, V, Flags, len);
}

V          = [5 7 3 1 4 2]
V(0)       = [1 1 1 1 0 0]
V = split(V, V(0)) = [4 2 5 7 3 1]
V(1)       = [0 1 0 1 1 0]
V = split(V, V(1)) = [4 5 1 2 7 3]
V(2)       = [1 1 0 0 1 0]
V = split(V, V(2)) = [1 2 3 4 5 7]

```

Figure 7: An example of the split radix sort on a vector containing three bit values. The $V(n)$ notation signifies extracting the n^{th} bit of each element.

```

p_inv_b(NotF, Flags, len);

plus_scan_i(I_down, NotF, len);
  /* number of 0s in Flags */
sum = NotF[len-1] + I_down[len-1];
  /* setup correct offset */
Flags[0] += sum;
plus_scan_i(I_up, Flags, len);
  /* restore flags */
Flags[0] -= sum;
  /* fix initial element of I_up */
I_up[0] = sum;
p_sel_i(Index, I_up, I_down, Flags, len);
perm_i(Result, V, Index, len);

V          = [5 7 3 1 4 2 7 2]
Flags      = [1 1 1 1 0 0 1 0]
NotF       = [0 0 0 0 1 1 0 1]
I_down     = [0 0 0 0 0 1 2 2]
I_up       = [3 4 5 6 6 6 7 7]
Index      = [3 4 5 6 0 1 7 2]
Result     = [4 2 2 5 7 3 1 7]

```

Figure 8: The split operation packs the elements with an 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. All lowercase variables are scalars and all capitalized variables are vectors.

```

subroutine split(dst, src, fl, len)
integer dst(*), src(*), fl(*)
integer len

index = 1
do 10 j = 1, len
  if (fl(j) .EQ. 0) then
    dst(index) = src(j)
    index = index + 1
10  endif

do 20 j = 1, len
  if (fl(j) .EQ. 1) then
    dst(index) = src(j)
    index = index + 1
20  endif

return
end

```

Figure 9: The split operation implemented in Fortran. The Fortran compiler did not vectorize this code.

Routine	l_c (clocks/element)	$n_{1/2}$ (elements)
split operation	10.07	665
Parallel radix sort (32 bits)	421.35	560
Parallel radix sort (64 bits)	896.83	523

Table 5: $l_c (= 1/r_\infty)$ and $n_{1/2}$ values for the split operation and the parallel radix sort routine. The numbers are for one processor of a Cray Y-MP.

4.2 Tree algorithms

This section describes three operations on trees:

- `branch_sums`: given a tree of values, returns to every node the sum of its children;
- `root_sums`: given a tree of values, returns to every node the sum of the values on the path from the node to the root;
- `delete_vertices`: given a tree, and a flag at each node, deletes all the vertices that have a flag set.

Figure 10 illustrates the effects of these operations. Although a serial implementation of the three operations is straightforward, vectorizing these operations is beyond the capability of any vectorizing compiler. The problem is that there is no way of turning the multiple recursive calls into a loop without introducing a stack to simulate the recursion, and this inhibits vectorization. To vectorize, we have to drastically alter both the representation of a tree and the algorithms used. A description of the representation and algorithms is beyond the scope of this paper. The interested reader is referred to [2]. The performance numbers

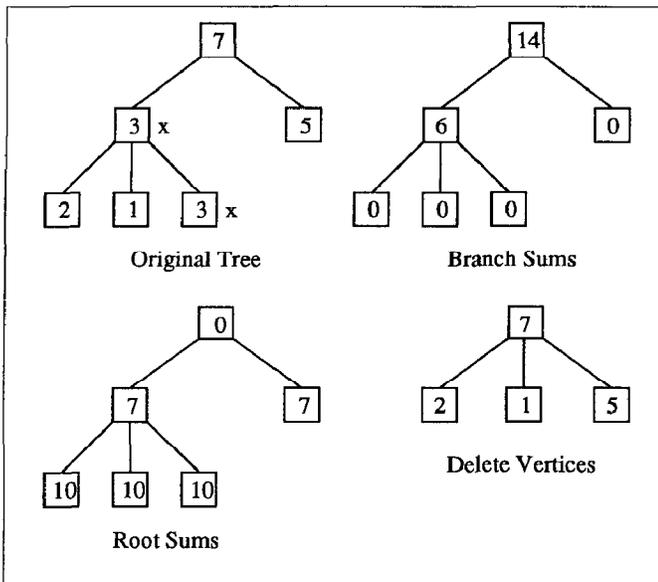


Figure 10: Illustration of the tree operations. For the *delete-vertices* the *x* marks the nodes to be deleted.

for the vector and serial versions of the operations are shown in Table 6.

The trees need not be binary or balanced for these operations. These basic operations can be used to execute the following useful operations on a tree:

- Determining the depth of each vertex. This is executed by applying a `root_sums` operation on a tree with the value 1 at every vertex.
- Determining how many descendants each vertex has. This is executed by applying a `branch_sums` operation on a tree with the value 1 at every vertex.
- Passing a flag down from a set of vertices to all their descendants. This is executed with an `root_sums` operation on a tree with a 1 in vertices that want their descendants marked. Any vertex that has a number greater than 0 as a result is marked.
- Passing a flag up from a set of leaves to all their ancestors. This is executed with an `branch_sums` operation on a tree with a 1 in vertices that want their ancestors marked. Any vertex that has a number greater than 0 as a result is marked.

4.3 A connectionist learning algorithm

As a final example that illustrates the use of the segmented versions of the scan primitives, we examine the forward propagation step of a connectionist learning algorithm called back-propagation. Connectionist network models are dynamic systems composed of a large number of simple processing units connected via weighted links, where the state of any unit in the network depends on the states of the units to which this unit connects. The values of the links or connections determine how

Routine	Vector Version		Scalar Version	
	l_e (clks/elt)	$n_{1/2}$ (elts)	l_e (clks/elt)	$n_{1/2}$ (elts)
<code>branch_sums</code>	12.2	489	206.5	2
<code>root_sums</code>	9.8	526	208.6	2
<code>delete_vertices</code>	19.2	483	276.2	2

Table 6: $l_e (= 1/v_{\infty})$ and $n_{1/2}$ values for the three tree operations. The numbers are for one processor of a Cray Y-MP.

one unit affects another. The overall behavior of the system is modified by adjusting the values of these connections through the repeated application of a learning rule.

The back-propagation learning algorithm is an error-correcting learning procedure for multilayered network architectures. Information flows forward through the network from an input layer, through intermediate, or hidden layer(s), to an output layer. The value of a unit is determined by first computing the weighted sum of the values of the units connected to this unit and then applying the logistic function, $\frac{1}{1+e^{-x}}$, to the result. This forward propagation rule is recursively applied to successively determine the unit values for each layer.

Forward propagation consists of four basic steps: distributing the activation values of the units to their respective fan-out weights, multiplying the activations by the weight values, summing these values from the weights into the next layer of units, and applying the logistic function to this value. Although dense matrix algorithms can be used to implement the steps if the network was very regular and dense, they will not work efficiently for sparse networks.

Given such a network represented as a directed graph, the forward propagation step proceeds as follows. First, the activations of the input units are clamped according to the input text. The algorithm then distributes the activations to the weights using a `copy-scan` operation, and then all weights multiply their weight by these activations. The result is sent from the output weights to the input weights of the unit in the next layer using a `permute`. A `plus-scan` then sums these input values into the next layer of units, and the logistic function is applied to the result at all the unit processors to determine the unit activations. This forward propagation step must be applied once for each layer of links in the network. This set of operations runs at 9.35 cycles/element.

5 Future work and conclusions

This paper has presented efficient implementations of both unsegmented and segmented scan primitives on vector machines, and shown how this leads to greatly improved performance on a representative set of applications. Each of the primitives has almost optimal asymptotic performance, but since they are made available to the application as a library of function calls, we cannot optimize across the primitives. Such optimizations could reduce temporary storage, allocate resources better, and reduce

redundant computation; this would make the split operation run twice as fast. We are working on compiler technology to address these issues. We are also looking at using multiple processors on the CRAY Y-MP to speed up the primitives themselves.

The demonstrated speeds of the primitives, and the prospect of even higher throughputs through the methods mentioned above, strongly argue for their inclusion in commonly-available programming libraries and languages. We believe that such scan primitives (including the segmented versions) should be easily accessible from and included in languages that support vector intrinsics (such as the new Fortran 8x specification). We hope that the ideas presented here will suggest new data structures and algorithmic techniques that will allow vector supercomputers to be used effectively on a much larger variety of problems.

References

- [1] American National Standards Institute. *American National Standard for Information Systems Programming Language Fortran: S8(X3.9-198x)*, March 1989.
- [2] Guy E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.
- [3] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.
- [4] Guy E. Blelloch and James J. Little. Parallel Solutions to Geometric Problems on the Scan Model of Computation. In *Proceedings International Conference on Parallel Processing*, pages Vol 3: 218–222, August 1988.
- [5] Guy E. Blelloch and Gary W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8(2), February 1990.
- [6] Cray Research Inc., Mendota Heights, Minnesota. *Symbolic Machine Instructions Reference Manual SR-0085B*, March 1988.
- [7] W. Daniel Hillis and Guy L. Steele Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12), December 1986.
- [8] R. W. Hockney. A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *Journal of the Association for Computing Machinery*, 12(1):95–113, January 1965.
- [9] R. W. Hockney and C. R. Jesshope. *Parallel Computers: Architecture, Programming, and Algorithms*. A. Hilger, Philadelphia, PA, Second Edition, 1988.
- [10] Kenneth E. Iverson. *A Programming Language*. Wiley, New York, 1962.
- [11] D. E. Knuth. *Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.
- [12] Peter M. Kogge and Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.
- [13] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.
- [14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.
- [15] John M. Levesque and Joel W. Williamson. *A Guidebook to Fortran on Supercomputers*. Academic Press, Inc., San Diego, CA, 1989.
- [16] James J. Little, Guy E. Blelloch, and Todd A. Cass. Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(3):244–257, March 1989.
- [17] Yu. Ofman. On the Algorithmic Complexity of Discrete Functions. *Soviet Physics Doklady*, 7(7):589–591, January 1963.
- [18] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Norwell, MA, 1988.
- [19] Harold S. Stone. Parallel Processing with the Perfect Shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, 1971.
- [20] Harold S. Stone. Parallel Tridiagonal Equation Solvers. *ACM Transactions on Mathematical Software*, 1(4):289–307, December 1975.
- [21] Y. Tanaka, K. Iwasawa, S. Gotoo, and Y. Umetani. Compiling Techniques for First-Order Linear Recurrences on a Vector Computer. In *Proceedings Supercomputing '88*, pages 174–181, Orlando, Florida, November 1988.
- [22] Hideo Wada, Koichi Ishiii, Masakazu Fukagawa, Hiroshi Murayama, and Shun Kawabe. High-speed Processing Schemes for Summation Type and Iteration Type Vector Instructions on HITACHI Supercomputer S-820 System. In *Proceedings 1988 ACM Conference on Supercomputing*, pages 197–206, July 1988.
- [23] Skef Wholey and Guy L. Steele Jr. Connection Machine Lisp: A Dialect of Common Lisp for Data Parallel Programming. In *Proceedings Second International Conference on Supercomputing*, May 1987.