Scans as Primitive Parallel Operations

GUY E. BLELLOCH

Abstract—**In most parallel random access machine (PRAM) models, memory references are assumed to take unit time. In practice, and in theory, certain scan operations, also known as prefix computations, can execute in no more time than these parallel memory references. This paper outlines an extensive study of the effect of including, in the PRAM models, such scan operations as unit-time primitives. The study concludes that the primitives improve the asymptotic running time of many algorithms by an $O(\log n)$ factor greatly simplify the description of many algorithms, and are significantly easier to implement than memory references. We therefore argue that the algorithm designer should feel free to use these operations as if they were as cheap as a memory reference.**

This paper describes five algorithms that clearly illustrate how the scan primitives can be used in algorithm design: a *radix-sort* algorithm, a *quicksort* algorithm, a *minimum-spanning-tree* algorithm, a *line-drawing* algorithm, and a *merging* algorithm. These all run on an EREW PRAM with the addition of two scan primitives, and are either simpler or more efficient than their pure PRAM counterparts.

The scan primitives have been implemented in microcode on the Connection Machine System, are available in PARIS (the parallel instruction set of the machine), and are used in a large number of applications. All five algorithms have been tested, and the radix sort is the currently supported sorting algorithm for the Connection Machine.

Index Terms—**Connection Machine, parallel algorithms, parallel computing, PRAM, prefix computations, scan.**

I. Introduction

ALGORITHMIC models typically supply a simple abstraction of a computing device and a set of primitive operations assumed to execute in a fixed "unit time." The assumption that primitives operate in unit time allows researchers to greatly simplify the analysis of algorithms, but is never strictly valid on real machines: primitives often execute in time dependent on machine and algorithm parameters. For example, in the serial random access machine (RAM) model [14], memory references are assumed to take unit time even though the data must fan-in on any real hardware and therefore take time that increases with the memory size. In spite of this inaccuracy in the model, the unit-time assumption has served as an excellent basis for the analysis of algorithms.

In the parallel random access machine (PRAM) models [16], [40], [42], [19], [20], memory references are again assumed to take unit time. In these parallel models, this "unit time" is large since there is no practical hardware known that does better than deterministic $O(\log^2 n)$, or probabilistic $O(\log n)$, bit times for an arbitrary memory reference from n processors.[1] This can lead to algorithms that are practical in the model but impractical on real machines. One solution is to use lower level models based on a fixed connectivity of the processors, such as the shuffle-exchange networks [44] or grid networks [47]. This, however, gives up machine independence and greatly complicates the description of algorithms. This paper suggests another solution: to add other primitives to the PRAM models that can execute as fast as memory references in practice, and that can reduce the number of program steps of algorithms—therefore making the algorithms more practical.

This paper outlines a study of the effect of including certain scan operations as such "unit time" primitives in the PRAM models. The *scan operations*[2] take a binary operator $\oplus$ with identity i, and an ordered set $[a_0, a_1, \cdots, a_{n-1}]$ of n elements, and returns the ordered set $[i, a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2})]$. In this paper, we consider two primitive scan operators, integer addition and integer maximum. These have a particularly simple implementation and they can be used to implement many other useful scan operations. On a PRAM, each element a_i is placed in a separate processor, and the scan executes over a fixed order of the processors—the prefix operation on a linked list [48], [27], and the fetch-and-op type instructions [21], [20], [37] are not considered. The conclusions of our study are summarized as follows.

- The scan primitives improve the asymptotic running time of many algorithms by an $O(\log n)$ factor over the EREW model and some by an $O(\log n)$ factor over the CRCW model (see Table I).
- The scan primitives simplify the description of many algorithms. Even in algorithms where the complexity is not changed from the pure PRAM model, the scan version is typically significantly simpler.
- Both in theory and in practice, the two scan operations can execute in less time than references to a shared memory, and can be implemented with less hardware (see Table II).

This paper is divided into two parts: *algorithms* and *implementation*. The first part illustrates how the scan primitives can be used in algorithm design, describes several interesting

Manuscript received March 20, 1987; revised March 16, 1989. This work was done within the Artificial Intelligence Laboratory, Massachusetts Institute of Technology. Support for the A.I. Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research under Contract N00014-85-K-0124.

The author is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 8930170.

[1] The AKS sorting network [1] takes $O(\log n)$ time deterministically, but is not practical.

[2] The Appendix gives a short history of the scan operations.

TABLE I
THE SCAN MODEL IS THE EREW PRAM MODE WITH THE ADDITION OF TWO SCAN PRIMITIVES. THESE SCAN PRIMITIVES IMPROVE THE ASYMPTOTIC RUNNING TIME OF MANY ALGORITHMS BY AN $O(\log n)$ FACTOR. THE ALGORITHMS NOT DESCRIBED IN THIS PAPER ARE DESCRIBED ELSEWHERE [7], [8]. SOME OF THE ALGORITHMS ARE PROBABILISTIC

	Model		
Algorithm	EREW	CRCW	Scan
Graph Algorithms (n vertices, m edges, m processors)			
Minimum Spanning Tree	$O(\log^2 n)$	$O(\log n)$	$O(\log n)$
Connected Components	$O(\log^2 n)$	$O(\log n)$	$O(\log n)$
Maximum Flow	$O(n^2 \log n)$	$O(n^2 \log n)$	$O(n^2)$
Maximal Independent Set	$O(\log^2 n)$	$O(\log^2 n)$	$O(\log n)$
Biconnected Components	$O(\log^2 n)$	$O(\log n)$	$O(\log n)$
Sorting and Merging (n keys, n processors)			
Sorting	$O(\log n)$	$O(\log n)$	$O(\log n)$
Merging	$O(\log n)$	$O(\log \log n)$	$O(\log \log n)$
Computational Geometry (n points, n processors)			
Convex Hull	$O(\log n)$	$O(\log n)$	$O(\log n)$
Building a K-D Tree	$O(\log^2 n)$	$O(\log^2 n)$	$O(\log n)$
Closest Pair in the Plane	$O(\log^2 n)$	$O(\log n \log \log n)$	$O(\log n)$
Line of Sight	$O(\log n)$	$O(\log n)$	$O(1)$
Matrix Manipulation ($n \times n$ matrix, n^2 processors)			
Matrix $\times$ Matrix	$O(n)$	$O(n)$	$O(n)$
Vector $\times$ Matrix	$O(\log n)$	$O(\log n)$	$O(1)$
Linear Systems Solver (with pivoting)	$O(n \log n)$	$O(n \log$ ..,	$O(n)$

TABLE II
BOTH IN THEORY AND IN PRACTICE CERTAIN SCAN OPERATIONS CAN EXECUTE IN LESS TIME THAN REFERENCES TO A SHARED MEMORY, AND CAN BE IMPLEMENTED WITH LESS HARDWARE. FOR THE CM-2 IMPLEMENTATION, THE SCAN IS IMPLEMENTED IN MICROCODE AND ONLY USES EXISTING HARDWARE

	Memory Reference	Scan Operation
Theoretical		
VLSI models		
Time	$O(\log n)$ [29]	$O(\log n)$ [30]
Area	$O(n^2/\log n)$	$O(n)$
Circuit models		
Depth	$O(\log n)$ [1]	$O(\log n)$ [15]
Size	$O(n \log n)$	$O(n)$
Actual		
64K processor CM-2		
Bit Cycles (Time)	600	550
Percent of Hardware	30	0

algorithms, and proves some of the results of Table I. The second part describes a very simple hardware implementation of the two scan primitives, and describes how other scan operations can be implemented on top of these two scans.

We call the exclusive-read exclusive-write (EREW) PRAM model with the scan operations included as primitives, the *scan* model. Since the term *unit time* is misleading (both memory references and scan operations take many clock cycles on a real machine), this paper henceforth uses the term *program step* or *step* instead. The number of program steps taken by an algorithm is the *step complexity*.

II. Algorithms

On first appearance, the scan operations might not seem to greatly enrich the memory reference operations of the PRAM

TABLE III
A CROSS REFERENCE OF THE VARIOUS USES OF SCANS INTRODUCED IN THIS PAPER WITH THE EXAMPLE ALGORITHMS DISCUSSED IN THIS PAPER. ALL THE USES CAN BE EXECUTED IN A CONSTANT NUMBER OF PROGRAM STEPS

Uses of Scan Primitives	Example Algorithms
Enumerating	Splitting, Load Balancing
Copying	Quicksort, Line Drawing, Minimum Spanning Tree
Distributing Sums	Quicksort, Minimum Spanning Tree
Splitting	Split Radix Sort, Quicksort
Segmented Primitives	Quicksort, Line Drawing, Minimum Spanning Tree
Allocating	Line Drawing, Halving Merge
Load-Balancing	Halving Merge

Example Algorithms	Uses of Scan Primitives
Split Radix Sort (2.2.1)	Splitting
Quicksort (2.3.1)	Splitting, Distributing Sums, Copying, Segmented Primitives
Minimum Spanning Tree (2.3.3)	Distributing Sums, Copying, Segmented Primitives
Line Drawing (2.4.1)	Allocating, Copying, Segmented Primitives
Halving Merge (2.5.1)	Allocating, Load Balancing

models. As we will discover in this paper, this is far from true; they are useful for a very broad set of algorithms. We separate the uses of scans into four categories. Section II-B, *simple operations*, shows how scans can be used to enumerate a subset of marked processors, to copy values across all processors, and to sum values across all processors. As an illustration of the use of enumerating, we describe a practical radix sort that requires $O(1)$ program steps for each bit of the keys. Section II-C, *segmented operations*, shows how segmented versions of the scans are useful in algorithms that work over many sets of data in parallel. As examples of the use of segmented scans, we describe a quicksort algorithm, which has an expected complexity of $O(\log n)$ program steps, and a minimum-spanning-tree algorithm with probabilistic complexity $O(\log n)$. Section II-D, *allocating*, shows how the scan operations are very useful for allocating processors. As an example, we describe a line-drawing algorithm which uses $O(1)$ program steps. Section II-E, *load-balancing*, shows how the scan operations are useful to load balance elements across processors when there are more data elements than processors. As an examples, we describe a merging algorithm which with p processors and n elements has a step complexity of $O(n/p + \log n)$.

Table III summarizes the uses of the scan operations and the example algorithms discussed in this paper. All the algorithms discussed in the paper have been implemented on the Connection Machine and in some of descriptions we mention the running times of the implementation. Before discussing the uses of the scan primitives, we introduce some notational conventions used in the paper.

A. Notation

We will assume that the data used by the algorithms in this paper are stored in vectors (one-dimensional arrays) in the shared memory and that each processor is assigned to one element of the vector. When executing an operation, the ith processor operates on the ith element of a vector. For example, in the operation

$$A = [5\quad 1\quad 3\quad 4\quad 3\quad 9\quad 2\quad 6]$$

$$B = [2\quad 5\quad 3\quad 8\quad 1\quad 3\quad 6\quad 2]$$

$$C \leftarrow A + B = [7\quad 6\quad 6\quad 12\quad 4\quad 12\quad 8\quad 8]$$

each processor reads its respective value from the vectors A and B, sums the values, and writes the result into the destination vector C. Initially, we assume that the PRAM always has as many processors as vector elements.

The *scan* primitives can be used to scan the elements of a vector. For example,

$$A = [2\quad 1\quad 2\quad 3\quad 5\quad 8\quad 13\quad 21]$$

$$C \leftarrow +\text{-scan}(A) = [0\quad 2\quad 3\quad 5\quad 8\quad 13\quad 21\quad 34].$$

In this paper, we use five primitive scan operations: **or-scan**, **and-scan**, **max-scan**, **min-scan**, and **+-scan**. We also use *backward* versions of each of these scans operations—versions that scan from the last element to the first. Section III-D shows that all the scans can be implemented with just two scans, a **max-scan** and a **+-scan**.

To reorder the elements of a vector, we use the **permute** operation. The **permute** operation, in the form **permute** (A, I), permutes the elements of A to the positions specified by the indexes of I. All indexes of I must be unique. For example,

$$A(\text{data vector}) = [a_0\quad a_1\quad a_2\quad a_3\quad a_4\quad a_5\quad a_6\quad a_7]$$

$$I(\text{index vector}) = [2\quad 5\quad 4\quad 3\quad 1\quad 6\quad 0\quad 7]$$

$$C \leftarrow \textbf{permute}(A, I) = [a_6\quad a_4\quad a_0\quad a_3\quad a_2\quad a_1\quad a_5\quad a_7].$$

To implement the **permute** operation on a EREW PRAM, each processor reads its respective *value* and *index*, and writes the *value* into the *index* position of the destination vector.

B. Simple Operations

We now consider three simple operations that are based on the scan primitives: enumerating, copying, and distributing sums (see Fig. 1). These operations are used extensively as part of the algorithms we discuss in this paper and all have an step complexity of $O(1)$. The **enumerate** operation returns the integer i to the ith true element. This operation is implemented by converting the flags to 0 or 1 and executing a **+-scan**. The **copy** operation copies the first element over all elements. This operation is implemented by placing the identity element in all but the first element of a vector and executing any scan.[3] Since the scan is not inclusive, we must put the first element back after executing the scan. The **+-distribute** operation returns to each element the sum of all the elements. This operation is implemented using a **+-scan** and a backward **copy**. We can likewise define a **max-distribute**, **min-distribute**, **or-distribute**, and **and-distribute**.

Flag	=	[T	F	F	T	F	T	T	F]
enumerate(Flag)	=	[$\boxed{0}$	1	1	$\boxed{1}$	2	$\boxed{2}$	$\boxed{3}$	4]
A	=	[5	1	3	4	3	9	2	6]
copy(A)	=	[5	5	5	5	5	5	5	5]
B	=	[1	1	2	1	1	2	1	1]
+-distribute(B)	=	[10	10	10	10	10	10	10	10]

Fig. 1. The **enumerate**, **copy**, and **+-distribute** operations. The **enumerate** numbers the flagged elements of a vector, the **copy** copies the first element across a vector, and the **+-distribute** sums the elements of a vector.

```
define split-radix-sort(A, number-of-bits){
    for i from 0 to (number-of-bits - 1)
        A ← split(A, A⟨i⟩)}
```

A	=	[5	7	3	1	4	2	7	2]
A⟨0⟩	=	[T	T	T	T	F	F	T	F]
A ← split(A, A⟨0⟩)	=	[4	2	2	5	7	3	1	7]
A⟨1⟩	=	[F	T	T	F	T	T	F	T]
A ← split(A, A⟨1⟩)	=	[4	5	1	2	2	7	3	7]
A⟨2⟩	=	[T	T	F	F	F	T	F	T]
A ← split(A, A⟨2⟩)	=	[1	2	2	3	4	5	7	7]

Fig. 2. An example of the split radix sort on a vector containing three bit values. The $A\langle n\rangle$ notation signifies extracting the nth bit of each element of the vector A and converting it to a Boolean value (T for 1, F for 0). The split operation packs F (0) elements to the bottom and T (1) elements to the top.

1) Example: Split Radix Sort: To illustrate the use of the scans for enumerating, consider a simple radix sorting algorithm. The algorithm is a parallel version of the standard serial radix sort [26].

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 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 since we iterate from the bottom bit up. Fig. 2 shows an example of the sort along with code to implement it.

We now consider how the **split** operation can be implemented in the scan model. The basic idea is to determine a new index for each element and permute the elements to these new indexes. To determine the new indexes for elements with

[3] One might think of defining a binary associative operator first which returns the first of its two arguments, and use it to execute the **copy** operation. The problem is that the first operator does not have an identity—a requirement for our definition of a scan.

```
define split(A, Flags){
    I-down ← enumerate(not(Flags));
    I-up ← n - back-enumerate(Flags) - 1;
    Index ← if Flags then I-up else I-down;
  permute(A, Index)}
```

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[T	T	T	T	F	F	T	F]
I-down	=	[0	0	0	0	$\boxed{0}$	$\boxed{1}$	2	$\boxed{2}$]
I-up	=	[$\boxed{3}$	$\boxed{4}$	$\boxed{5}$	$\boxed{6}$	6	6	$\boxed{7}$	7]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]

Fig. 3. The **split** operation packs the elements with an F in the corresponding flag position to the bottom of a vector, and packs the elements with a T to the top of the same vector.

TABLE IV
COMPARISON OF THE TIMES TAKEN BY THE SPLIT RADIX SORT AND THE BITONIC SORT (n KEYS EACH WITH d BITS). THE CONSTANTS IN THE THEORETICAL TIMES ARE VERY SMALL FOR BOTH ALGORITHMS. ON THE CONNECTION MACHINE, THE BITONIC SORT IS IMPLEMENTED IN MICROCODE WHEREAS THE SPLIT RADIX SORT IS IMPLEMENTED IN MACROCODE, GIVING THE BITONIC SORT AN EDGE

	Split Radix Sort	Bitonic Sort
Theoretical (Bit Serial Circuit) Bit Time	$O(d \log n)$	$O(d + \log^2 n)$
Actual (64K processor CM-1) Bit cycles (sorting 16 bits)	20,000	19,000

a 0 (F) in the bit, we enumerate these elements as described in the last section. To determine the new indexes of elements with a 1 (T) in the bit, we enumerate the elements starting at the top of the vector and subtract these from the length of the vector. Fig. 3 shows an example of the **split** operation along with code to implement it.

The **split** operation has a step complexity of $O(1)$; so for d-bit keys, the split radix sort has a step complexity of $O(d)$. If we assume for n keys that the keys are $O(\log n)$ bits long, a common assumption in models of computation [45], then the algorithm has a step complexity of $O(\log n)$. Although $O(\log n)$ is the same asymptotic complexity as existing EREW and CRCW algorithms [1], [11], the algorithm is much simpler and has a significantly smaller constant. Note that since integers, characters, and floating-point numbers can all be sorted with a radix sort, a radix sort suffices for almost all sorting of fixed-length keys required in practice.

The split radix sort is fast in the scan model, but is it fast in practice? After all, our architectural justification claimed that the scan primitives bring the P-RAM models closer to reality. Table IV compares implementations of the split radix sort and Batcher's bitonic sort [4] on the Connection Machine. We choose the bitonic sort for comparison because it is commonly cited as the most practical parallel sorting algorithm. I have also looked into implementing Cole's sort [11], which is optimal on the P-RAM models, on the Connection Machine. Although never implemented, because of its complexity, it was estimated that it would be at least a factor of 4, and possibly a factor of 10, slower than the other two sorts. The split radix sort is the sort currently supported by the parallel instruction set of the Connection Machine [46].

C. Segments and Segmented Scans

In many algorithms, it is useful to break the linear ordering of the processors into segments and have a scan operation start again at the beginning of each segment; we call such scan operations, *segmented scans*. Segmented scans take two arguments: a set of values and a set of segment flags. Each flag in the segment flags specifies the start of a new segment (see Fig. 4). Segmented scans were first suggested by Schwartz [40] and this paper shows some useful applications of these scans.

The segmented scan operations are useful because they allow algorithms to execute the scans independently over the elements of many sets. This section discusses how they can be used to implement a parallel version of quicksort and how they can be used to represent graphs. This graph representation is then used in a minimum-spanning-tree algorithm.

The segmented scan operations can all be implemented with at most two calls to the two unsegmented primitive scans (see Section III-D). They can also be implemented directly as described by Schwartz [40].

1) Example: Quicksort: To illustrate the use of segments, we consider a parallel version of Quicksort. Similar to the standard serial version [24], the parallel version picks one of the keys as a pivot value, splits the keys into three sets—keys lesser, equal, and greater than the pivot—and recurses on each set.[4] The algorithm has an expected step complexity of $O(\log n)$.

The basic intuition of the parallel version is to keep each subset in its own segment, and to pick pivot values and split the keys independently within each segment (see Fig. 5). The steps required by the sort are outlined as follows:

1. Check if the keys are sorted and exit the routine if they are. Each processor checks to see if the previous processor has a lesser or equal value. We execute an **and-distribute** (Section II-B) on the result of the check so that each processor knows whether all other processors are in order.

2. Within each segment, pick a pivot and distribute it to the other elements.

If we pick the first element as a pivot, we can use a segmented version of the **copy** (Section II-B) operation implemented based on a segmented **max-scan**. The algorithm could also pick a random element by generating a random number in the first element of each segment, moding it with the length of the segment, and picking out the element with a few scans.

3. Within each segment, compare each element to the pivot and split based on the result of the comparison.

For the split, we can use a version of the **split** operation described in Section II-B1 which splits into three sets instead of two, and which is segmented. To implement such a segmented **split**, we can use a segmented version of the **enumerate** operation (Section II-B) to number relative to the beginning of

[4] We do not need to recursively sort the keys equal to the pivot, but the algorithm as described below does.

A	=	[5	1	3	4	3	9	2	6]
Sb	=	[T	F	T	F	F	F	T	F]
seg-+-scan(A, Sb)	=	[0	5	0	3	7	10	0	2]
seg-max-scan(A, Sb)	=	[0	5	0	3	4	4	0	2]

Fig. 4. The segmented scan operations restart at the beginning of each segment. The vector *Sb* contains flags that mark the beginning of the segments.

Key	=	[6.4	9.2	3.4	1.6	8.7	4.1	9.2	3.4]
Segment-Flags	=	[T	F	F	F	F	F	F	F]
Pivots	=	[6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4]
F	=	[=	>	<	<	>	<	>	<]
Key ← **split**(Key, F)	=	[3.4	1.6	4.1	3.4	6.4	9.2	8.7	9.2]
Segment-Flags	=	[T	F	F	F	T	T	F	F]
Pivots	=	[3.4	3.4	3.4	3.4	6.4	9.2	9.2	9.2]
F	=	[=	<	>	=	=	=	<	=]
Key ← **split**(Key, F)	=	[1.6	3.4	3.4	4.1	6.4	8.7	9.2	9.2]
Segment-Flags	=	[T	T	F	T	T	T	T	F]

Fig. 5. Parallel quicksort. On each step, within each segment, we distribute the pivot, test whether each element is equal-to, less-than, or greater-than the pivot, split into three groups, and generate a new set of segment flags.

each segment, and we can use a segmented version of the **copy** operation to copy the offset of the beginning of each segment across the segment. We then add the offset to the numbers relative to beginning of the segment to generate actual indexes to which we permute each element.

4. Within each segment, insert additional segment flags to separate the split values. Knowing the pivot value, each element can determine if it is at the beginning of the segment by looking at the previous element.

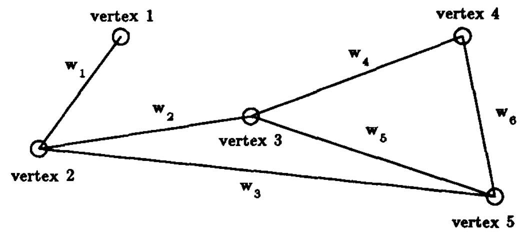
5. Return to step 1.

Each iteration of this sort requires a constant number of calls to the primitives. If we select pivots randomly within each segment, quicksort is expected to complete in $O(\log n)$ iterations, and therefore has an expected step complexity of $O(\log n)$. This version of quicksort has been implemented on the Connection Machine and executes in about twice the time as the split radix sort.

The technique of recursively breaking segments into subsegments and operating independently within each segment can be used for many other divide-and-conquer algorithms [7], [8].

2) Graphs: An undirected graph can be represented using a segment for each vertex and an element position within a segment for each edge of the vertex. Since each edge is incident on two vertices, it appears in two segments. The actual values kept in the elements of the segmented vector are pointers to the other end of the edge (see Fig. 6). To include weights on the edges of the graphs, we can use an additional vector that contains the weights of the edges.

By using segmented operations to operate over the edges of each vertex, the step complexity of many useful operations on graphs can be reduced. For example, for n vertices, the



Index	=	[0	1	2	3	4	5	6	7	8	9	10	11]
vertex	=	[1	2	2	2	3	3	3	4	4	5	5	5]
segment-descriptor	=	[T	T	F	F	T	F	F	T	F	T	F	F]
cross-pointers	=	[1	0	4	$\boxed{9}$	2	7	10	5	11	$\boxed{3}$	6	8]
weights	=	[w_1	w_1	w_2	w_3	w_2	w_4	w_5	w_4	w_6	w_3	w_5	w_6]

Fig. 6. An example of the undirected segmented graph representation. Each pointer points to the other end of the edge. So, for example, edge w_3 in vertex 2 contains a pointer (in this case 9) to its other end in vertex 5. The segment descriptor partitions the processors into vertices.

complexity of each vertex summing a value from all neighbors is reduced from $O(\log n)$ in the PRAM models to $O(1)$ in the scan model. Such neighbor summing can be executed by distributing the value from each vertex over its edges using a segmented **copy** operation, permuting these values using the cross-pointers, and summing the values on the edges back into the vertices using a segmented **+-distribute** operation. Elsewhere [7] we show that by keeping trees in a particular form, we can similarly reduce the step complexity of many tree operations on trees with n vertices by $O(\log n)$.

For this segmented graph representation to be useful, either there must be an efficient routine to generate the representation from another representation, or it must be possible to do all manipulations on graphs using the segmented graph representation. A graph can be converted from most other representations into the segmented graph representation by creating two elements per edge (one for each end) and sorting the edges according to their vertex number. The split radix sort (Section II-B1) can be used since the vertex numbers are all integers less than n. The sort places all edges that belong to the same vertex in a contiguous segment. We suggest, however, that in the scan model graphs always be kept in the segmented graph representation.

3) Minimum Spanning Tree: This section describes a probabilistic minimum-spanning-tree (MST) algorithm. For n vertices and m edges, it has a step complexity of $O(\log n)$. The best algorithm known for the EREW PRAM model requires $O(\log^2 n)$ time [23], [39]. The best algorithm known for the CRCW PRAM model requires $O(\log n)$ time [3], but this algorithm requires that the generic CRCW PRAM model be extended so that if several processors write to the same location, either the value from the lowest numbered processor is written or the minimum value is written.

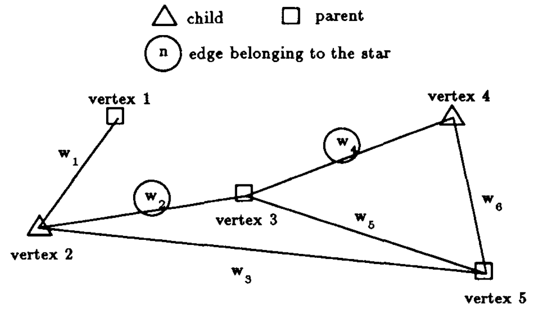
All these algorithms are based on the algorithm of Sollin [5], which is similar to the algorithm of Borůvka [9]. The algorithms start with a forest of trees in which each tree is a single vertex. These trees are merged during the algorithm, and the algorithm terminates when a single tree remains. At each step, every tree T finds its minimum-weight edge joining a vertex in T to a vertex of a distinct tree T'. Trees connected by one of these edges merge. To reduce the forest to a single tree, $O(\log n)$ such steps are required.

In the EREW PRAM algorithm, each step requires $\Omega(\log n)$ time because finding the minimum edge in a tree and distributing connectivity information over merging trees might require $\Omega(\log n)$ time. In the extended CRCW PRAM model, each step only requires constant time because each minimum edge can be found with a single write operation. In our algorithm, we keep the graph in the graph representation discussed in Section II-C2 so that we can use the **min-distribute** (Section II-B) operation to find the minimum edge for each tree and the **copy** operation to distribute connectivity information among merging trees with a constant number of calls to the primitives. The only complication is maintaining the representation when merging trees.

As with the Shiloach and Vishkin CRCW PRAM algorithm [43], trees are selected for merging by forming stars. We define a star as a set of vertices within a graph with one of the set marked as the parent, the others marked as children, and an edge that leads from each child vertex to its parent vertex.[5] A graph might contain many stars. The *star-merge* operation takes a graph with a set of disjoint stars, and returns a graph with each star merged into a single vertex. Fig. 7 shows an example of a star-merge for a graph with a single star.

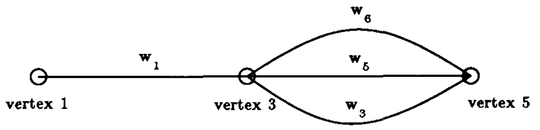
The minimum-spanning-tree algorithm consists of repeatedly finding stars and merging them. To find stars, each vertex flips a coin to decide whether they are a child or parent. All children find their minimum edge (using a **min-distribute**),

[5] This definition of a star is slightly different from the definition of Shiloach and Vishkin [43].



Before Star-Merge

Index	=	[0	1	2	3	4	5	6	7	8	9	10	11]
segment-descriptor	=	[T	T	F	F	T	F	F	T	F	T	F	F]
weights	=	[w_1	w_1	w_2	w_3	w_2	w_4	w_5	w_4	w_6	w_3	w_5	w_6]
Star-Edge	=	[F	F	T	F	T	T	F	T	F	F	F	F]
Parent	=	[T	F			T			F		T]		



After Star-Merge

Index	=	[0	1	2	3	4	5	6	7]
segment-descriptor	=	[T	T	F	F	F	T	F	F]
weights	=	[w_1	w_1	w_3	w_5	w_6	w_3	w_5	w_6]
cross-pointers	=	[1	0	5	6	7	2	3	4]

Fig. 7. An example of star merging. In this example, two parents have no children (1 and 5) and the third (3) has two children (2 and 4). The second diagram shows the graph after the star is merged.

and all such edges that are connected to a parent are marked as *star edges*. Since, on average, half the trees are children and half of the trees on the other end of the minimum edge of a child are parents, 1/4 of the trees are merged on each star-merge step. This random mate technique is similar to the method discussed by Miller and Reif [33]. Since, on average, 1/4 of the trees are deleted on each step, $O(\log n)$ steps are required to reduce the forest to a single tree.

We now describe how a star-merge operation can be implemented in the scan model, such that for m edges, the operation has a step complexity of $O(1)$. The input to the star-merge operation is a graph in the segmented graph representation, with two additional vectors: one contains flags that mark every star edge, and the other contains a flag that marks every parent. To implement a star-merge and maintain the segmented graph representation, each child segment must be moved into its parent segment. The technique used for this rearrangement can be partitioned into four steps: 1) each parent opens enough space in its segment to fit its children, 2) the children are permuted into this space, 3) the cross-pointers vector is updated to reflect the change in structure of the graph, and 4) edges which point within a segment are deleted, therefore deleting edges that point within a tree.

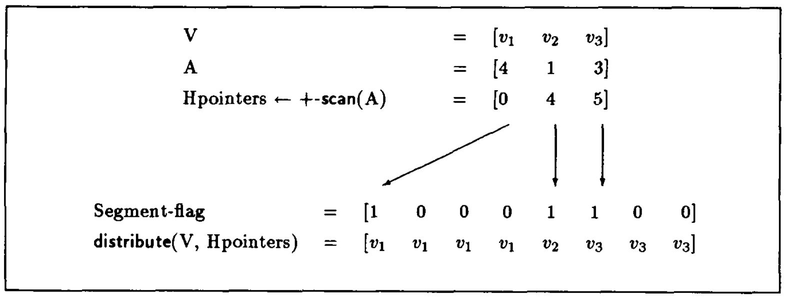

V = [v1 v2 v3]
A = [4 1 3]
Hpointers ← +-scan(A) = [0 4 5]
Segment-flag = [1 0 0 0 1 1 0 0]
distribute(V, Hpointers) = [v1 v1 v1 v1 v2 v3 v3 v3]

Fig. 8. Processor allocation. The vector A specifies how many new elements each position needs. We can allocate a segment to each position by applying a +-**scan** to A and using the result as pointers to the beginning of each segment. We can then distribute the values of V to the new elements with a permute to the beginning of the segment and a segmented copy across the segment.

1) To open space in the parent segments, each child passes its length (number of edges) across its star edge to its parent, so each parent knows how much space it needs to open up for each of its children. Let us call the vector that contains the needed space of each child the *needed space* vector. A 1 is placed in all the nonstar edges of this vector. We can now use a segmented +-**distribute** on the *needed-space* vector to determine the new size of each parent and a +-**scan** to allocate this new space for each parent (such allocation is discussed in more detail in Section II-D). We also execute a segmented +-**scan** on the *needed-space* vector to determine the offset of each child within its parent segment and the new position of each nonstar edge of the parent. We call this vector the *child-offset* vector.

2) We now need to permute the children into the parent segments. To determine the new position of the edges in the child vertices, we permute the *child-offset* back to each child and distribute it across the edges of the child. Each child adds its index to this offset giving each child edge a unique address within the segment of its parent. We now permute all the edges, children and parents, to their new positions. 3) To update the pointers, we simply pass the new position of each end of an edge to the other end of the edge. 4) To delete edges that point within a segment, we check if each edge points within the segment by distributing the ends of the segment, and then pack all elements that point outside each segment deleting elements pointing within each segment. The pointers are updated again.

D. Allocating

This section illustrates another use of the scan operations. Consider the problem of given a set of processors each with an integer, allocating that integer number of new processors to each initial processor. Such allocation is necessary in the parallel line-drawing routine described in Section II-D1. In the line-drawing routine, each line calculates the number of pixels in the line and dynamically allocates a processor for each pixel. Allocating new elements is also useful for the branching part of many branch-and-bound algorithms. Consider, for example, a brute force chess-playing algorithm that executes a fixed-depth search of possible moves to determine the best next move.[6] We can execute the algorithms in parallel by placing each possible move in a separate processor. Since the algorithm dynamically decides how many next moves to generate, depending on the position, we need to dynamically allocate new elements. In Section II-E, we discuss the bounding part of branch-and-bound algorithms.

Defined more formally, allocation is the task of, given a vector of integers A with elements a_i and length l, creating a new vector B of length

$$L = \sum_{i=0}^{l-1} a_i \qquad (1)$$

with a_i elements of B assigned to each position i of A. By assigned to, we mean that there must be some method for distributing a value at position i of a vector to the a_i elements which are assigned to that position. Since there is a one-to-one correspondence between elements of a vector and processors, the original vector requires l processors and the new vector requires L processors. Typically, an algorithm does not operate on the two vectors at the same time, so the processors can overlap.

Allocation can be implemented by assigning a contiguous segment of elements to each position i of A. To allocate segments we execute a +-**scan** on the vector A returning a pointer to the start of each segment (see Fig. 8). We can then generate the appropriate segment flags by permuting a flag to the index specified by the pointer. To distribute values from each position i to its segment, we permute the values to the beginning of the segments and use a segmented **copy** operation to copy the values across the segment. Allocation and distribution each require $O(1)$ steps on the scan model. Allocation requires $O(\log n)$ steps on a EREW P-RAM and $O(\log n/\log \log n)$ steps on a CREW P-RAM (this is based on the prefix sum routine of Cole and Vishkin [13]).

When an algorithm allocates processors, the number of processors required is usually determined dynamically and will depend on the data. To account for this, we must do one of

[6] This is how many chess playing algorithms work [6]. The search is called a *minimax* search since it alternates moves between the two players, trying to minimize the benefit of one player and maximize the benefit of the other.

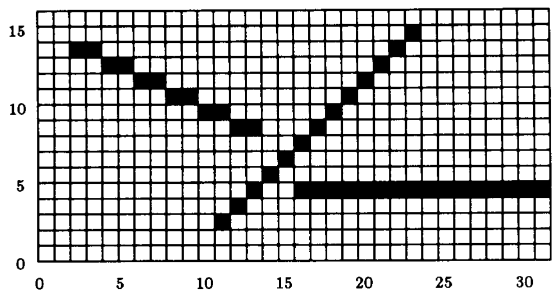


Fig. 9. The pixels generated by a line drawing routine. In this example, the endpoints are (11, 2)-(23, 14), (2, 13)-(13, 8), and (16, 4)-(31, 4). The algorithm allocates 12, 11, and 16 pixels, respectively, for the three lines.

$$[\underbrace{4\ \ 7\ \ 1}_{\text{processor 0}}\quad \underbrace{0\ \ 5\ \ 2}_{\text{processor 1}}\quad \underbrace{6\ \ 4\ \ 8}_{\text{processor 2}}\quad \underbrace{1\ \ 9\ \ 5}_{\text{processor 3}}]$$

```
Sum            =  [12   7   18   15]
+-scan(Sum)    =  [0    12  19   37]
```

$$[\underbrace{0\ \ 4\ \ 11}_{\text{processor 0}}\quad \underbrace{12\ \ 12\ \ 17}_{\text{processor 1}}\quad \underbrace{19\ \ 25\ \ 29}_{\text{processor 2}}\quad \underbrace{37\ \ 38\ \ 47}_{\text{processor 3}}]$$

Fig. 10. When operating on vectors with more elements than processors (*long vectors*), each processor is assigned to a contiguous block of elements. To execute an arithmetic operation or the permute operation on a long vector, each processor loops over the element positions for which it is responsible and executes the operation. To execute a scan, each processor sums its elements, executes a scan across processors, and uses the result as an offset to scan within processors.

TABLE V
THE PROCESSOR-STEP COMPLEXITY OF MANY ALGORITHMS CAN BE REDUCED BY USING FEWER PROCESSORS AND ASSIGNING MANY ELEMENTS TO EACH PROCESSOR

	Processors	Steps	Processor-Step
Halving Merge	$O(n)$	$O(\log n)$	$O(n \log n)$
	$O(n/\log n)$	$O(\log n)$	$O(n)$
List Ranking [12]	$O(n)$	$O(\log n)$	$O(n \log n)$
	$O(n/\log n)$	$O(\log n)$	$O(n)$
Tree Contraction [18]	$O(n)$	$O(\log n)$	$O(n \log n)$
	$O(n/\log n)$	$O(\log n)$	$O(n)$

three things: assume an infinite number of processors, put a bound on the number of elements that can be allocated, or start simulating multiple elements on each processor. The first is not practical, and the second restricting. Section II-E discusses the third.

1) Example: *Line Drawing*: As a concrete example of how allocation is used, consider line drawing. Line drawing is the problem of, given a set of pairs of points (each point is an (x, y) pair), generating all the locations of pixels that lie between one of the pairs of points. Fig. 9 illustrates an example. The routine we discuss returns a vector of (x, y) pairs that specify the position of each pixel along the line. It generates the same set of pixels as generated by the simple digital differential analyzer (DDA) serial technique [34].

The basic idea of the routine is for each line to allocate a processor for each pixel in the line, and then for each allocated pixel to determine, in parallel, its final position in the grid. To allocate a processor for each pixel, each line must first determine the number of pixels in the line. This number can be calculated by taking the maximum of the x and y differences of the line's endpoints. Each line now allocates a segment of processors for its pixels, and distributes its endpoints across the segment as described earlier. We now have one processor for each pixel and one segment for each line. We can view the position of a processor in its segment as the position of a pixel in its line. Based on the endpoints of the line and the position in the line (determined with a +-**scan**), each pixel can determine its final (x, y) location in the grid [34]. To actually place the points on a grid, we would need to permute a flag to a position based on the location of the point. In general, this will require the simplest form of concurrent-write (one of the values gets written) since a pixel might appear in more than one line.

This routine has a step complexity of $O(1)$ and requires as many processors as pixels in the lines. The routine has been implemented on the Connection Machine, has been extended to render solid objects by Salem, and is part of a rendering package for the Connection Machine [38].

E. Load Balancing

Up to now, this paper has assumed that a PRAM always has as many processors as elements in the data vectors. This section considers simulating multiple elements on each processor. Such simulation is important for two reasons. First, from a practical point of view, real machines have a fixed number of processors but problem sizes vary: we would rather not restrict ourselves to fixed, and perhaps small, sized problems. Second, from both a practical and theoretical point of view, by placing multiple elements on each processor, an algorithm can more efficiently utilize the processors and can greatly reduce the processor-step complexity (see Table V). Fig. 10 discusses how to simulate the various vector operations discussed in Section II-A on vectors with more elements than processors.

When simulating multiple elements on each processor, it is important to keep the number of elements on the processors balanced. We call such balancing, *load balancing*. Load balancing is important when data elements drop out during the execution of an algorithm since this might leave the remaining elements unbalanced. There are three common reasons why elements might drop out. First, some elements might have completed their desired calculations. For example, in the quicksort algorithm described in Section II-C1, segments which are already sorted might drop out. Second, the algorithm might be subselecting elements. Subselection is used in the halving merge algorithm discussed in Section II-E1. Third, an algorithm might be pruning some sort of search. Pruning might be used in the bounding part of branch-and-bound algorithms such as the chess-playing algorithm we mentioned in Section II-D. In all three cases, when the elements drop out, the number of elements left on each processor might be unbalanced.

For m remaining elements, load balancing can be imple-

$$\begin{aligned} F &= [T\ \ F\ \ F\ \ F\ \ T\ \ T\ \ F\ \ T\ \ T\ \ T\ \ T\ \ T] \\ A &= [\underbrace{a_0\ \ a_1\ \ a_2}_{\text{processor 0}}\ \ \underbrace{a_3\ \ a_4\ \ a_5}_{\text{processor 1}}\ \ \underbrace{a_6\ \ a_7\ \ a_8}_{\text{processor 2}}\ \ \underbrace{a_9\ \ a_{10}\ \ a_{11}}_{\text{processor 3}}] \end{aligned}$$

$$A = [\underbrace{a_0\ \ a_4}_{\text{proc 0}}\ \ \underbrace{a_5\ \ a_7}_{\text{proc 1}}\ \ \underbrace{a_8\ \ a_9}_{\text{proc 2}}\ \ \underbrace{a_{10}\ \ a_{11}}_{\text{proc 3}}]$$

Fig. 11. Load balancing. In load balancing, certain marked elements are dropped and the remaining elements need to be balanced across the processors. Load balancing can be executed by packing the remaining elements into a smaller vector using an enumerate and a permute, and assigning each processor to a smaller block.

mented by enumerating the remaining elements, permuting them into a vector of length m, and assigning each processor to m/p elements of the new vector (see Fig. 11). We call the operation of packing elements into a smaller vector, the *pack* operation. A processor can determine how many and which elements it is responsible for simply by knowing its processor number and m; m can be distributed to all the processors with a **copy**. In the scan model, load balancing requires $O(n/p)$ steps. On an EREW PRAM, load balancing requires $O(n/p + \log n)$ steps.

1) Example: Halving Merge: To illustrate the importance of simulating multiple elements on each processor and load balancing, this section describes an algorithm for merging two sorted vectors.[7] We call the algorithm, the *halving merge*. When applied to vectors of length n and m ($n \geq m$) on the scan model with p processors, the halving merge algorithm has a step complexity of $O(n/p + \log n)$. When $p < n/\log n$, the algorithm is optimal. Although the split radix sort and the quicksort algorithms are variations of well-known algorithms translated to a new model, the merging algorithm described here is original. The merging algorithm of Shiloach and Vishkin for the CRCW PRAM model [17], [42] has the same complexity but is quite different. Their algorithm is not recursive.

The basic idea of the halving merge algorithm is to extract the odd-indexed elements from each of the two vectors by packing them into smaller vectors, to recursively merge the half-length vectors, and then to use the result of the halving merge to determine the positions of the even-indexed elements. The number of elements halves on each recursive call, and the recursion completes when one of the merge vectors contains a single element. We call the operation of taking the result of the recursive merge on the odd-indexed elements and using it to determine the position of the even-indexed elements, *even-insertion*. We first analyze the complexity of the halving merge assuming that the even-insertion requires a constant number of scan and permute operations, and then discuss the algorithm in more detail.

The complexity of the algorithm is calculated as follows. Since the number of elements halves at each level, there are at most $\log n$ levels and at level i, $n/2^i$ elements must be merged. With p processors, if we load balance, the most elements any

$$\begin{array}{lcl} \text{A} & = & [1\ \ 7\ \ 10\ \ 13\ \ 15\ \ 20] \\ \text{B} & = & [3\ \ 4\ \ 9\ \ 22\ \ 23\ \ 26] \\ \text{A}' & = & [1\ \ 10\ \ 15] \\ \text{B}' & = & [3\ \ 9\ \ 23] \\ \text{merge(A}', \text{B}') & = & [1\ \ 3\ \ 9\ \ 10\ \ 15\ \ 23] \\ \text{near-merge} & = & [1\ \ \boxed{7\ \ 3\ \ 4}\ \ 9\ \ \boxed{22\ \ 10\ \ 13\ \ 15\ \ 20}\ \ 23\ \ 26] \\ \text{result} & = & [1\ \ 3\ \ 4\ \ 7\ \ 9\ \ 10\ \ 13\ \ 15\ \ 20\ \ 22\ \ 23\ \ 26] \end{array}$$

Fig. 12. The halving merge involves selecting the odd-indexed elements of each vector to be merged, recursively merging these elements and then using the result to merge the even-indexed elements (even-insertion). To execute the even-insertion, we place the even-indexed elements in the merged odd-indexed elements after their original predecessor. This vector, the *near-merge* vector, is almost sorted. As shown in the figure, nonoverlapping blocks might need to be rotated: the first element moved to the end.

processor is responsible for is

$$\lceil n/2^i p \rceil. \qquad (2)$$

If the even insertion requires a constant number of calls to the primitives per element, level i has a step complexity of

$$O(\lceil n/2^i p \rceil). \qquad (3)$$

The total step complexity is therefore

$$O\left(\sum_{i=0}^{\log n - 1} \left\lceil \frac{n}{2^i p} \right\rceil\right) = O\left(\sum_{i=0}^{\log n - 1} \left(\frac{n}{2^i p} + 1\right)\right) = O(n/p + \log n). \qquad (4)$$

We now discuss the algorithm in more detail. Picking every other element before calling the algorithm recursively can be implemented by marking the odd-indexed elements and packing them (load balancing them). After the recursive call returns, the even-insertion is executed as follows. We expand the merged odd-indexed vector by a factor of two by placing each unmerged even-indexed element directly after the element it originally followed (see Fig. 12). We call this vector the *near-merge* vector. The *near-merge* vector has an interesting property: elements can only be out of order by single nonoverlapping rotations. An element might appear before a block of elements it belongs after. We call such an element a block-head. A near-merge vector can be converted into a true merged vector by moving the block-head to the end of the block and sliding the other elements down by one: rotating the block by one. The rotation of the blocks can be implemented with two scans and two arithmetic operations:

```
define fix-near-merge(near-merge){
    head-copy←max(max-scan(near-merge), near-merge)
    result←min(min-backscan(near-merge), head-copy)}.
```

The first step moves the block-head to the end of the block, and the second step shifts the rest of the block down by one. The even-insertion therefore requires a constant number of calls to the vector operations.

To place the even-indexed elements following the odd-

[7] In this algorithm, the elements drop out in a very regular fashion and the remaining elements could be balanced without the scan primitives. Load balancing using scans, however, simplifies the description.

indexed elements after returning from the recursive call, we must somehow know the original position of each merged odd-indexed element. To specify these positions, the merge routine could instead of returning the actual merged values, return a vector of flags: each F flag represents an element of A and each T flag represents an element of B. For example,

$$A' = [1 \quad 10 \quad 15]$$

$$B' = [3 \quad 9 \quad 23]$$

$$\text{halving-merge}(A', B') = [F \quad T \quad T \quad F \quad F \quad T]$$

which corresponds to the merged values:

$$[1 \quad 3 \quad 9 \quad 10 \quad 15 \quad 23].$$

The vector of flags—henceforth the *merge-flag* vector—both uniquely specifies how the elements should be merged and specifies in which position each element belongs.

III. Implementation

This section describes a circuit that implements two scan primitives, integer versions of the +-**scan** and **max-scan**, and describes how the other scans used in this paper can be simulated with the two primitives. From a theoretical orientation, efficient circuits for implementing scan primitives have been discussed elsewhere [28], [15]. This section therefore concentrates on a practical implementation, described at the logic level, and discusses how this implementation could fit into an actual machine. Elsewhere we have shown [7] that some of the other scan operations, such as the segmented scan operations, can be implemented directly with little additional hardware.

Although the discussion suggests a separate circuit (set of chips) to implement the scan operations, wires and chips of a scan circuit might be shared with other circuitry. The scan implementation on the Connection Machine, for example, shares the wires with the router and requires no additional hardware.

A. Tree Scan

Before describing details on how a circuit is implemented, we review a standard, general technique for implementing the scan operation on a balanced binary tree for any binary associative scan operator $\oplus$. The technique consists of two sweeps of the tree, an up sweep and a down sweep, and requires 2 log n steps. Fig. 13 shows an example. The values to be scanned start at the leaves of the tree. On the up sweep, each unit executes $\oplus$ on its two children units and passes the sum to its parent. Each unit also keeps a copy of the value from the left child in its memory. On the down sweep, each unit passes to its left child the value from its parent and passes to its right child $\oplus$ applied to its parent and the value stored in the memory (this value originally came from the left child). After the down sweep, the values at the leaves are the results of a scan.

If the scan operator $\oplus$ can be executed with a single pass over the bits of its operand, such as integer addition and integer maximum, the tree algorithm can be bit pipelined. Bit pipelining involves passing the operands one bit at a time up

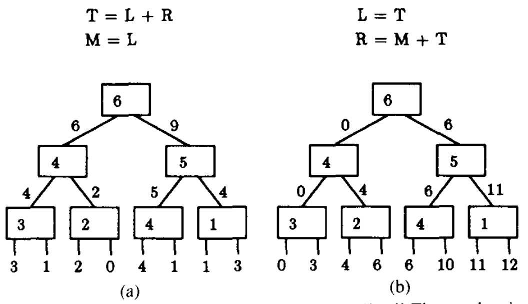


Fig. 13. Parallel scan on a tree using the operator "+." The number in the block is the number being stored in the memory on the up sweep. (a) After up sweep. (b) After down sweep.

the tree so that when the second level is working on bit n, the first level works on bit $n + 1$. Such bit pipelining can greatly reduce the hardware necessary to implement the scan operations since only single bit logic is required at each unit.

As an example of how bit pipelining works, we consider a bit-pipelined version of +-**scan** for n, m bit integers. This bit-pipelined scan starts by passing the least significant bit of each value into the leaf units of the tree. Each unit now performs a single bit addition on its two input bits, stores the carry bit in a flag, propagates the sum bit to its parent in the next layer of units, and stores the bit from the left child in an m bit memory on the unit. On the second step, the scan passes the second bit of each value into the leaf units of the tree while it propagates the least significant bit of the sums on the first layer to the second layer. In general, on the ith step, at the jth layer (counting from the leaves), the $(i - j)$th bit of the sum of a unit (counting from the least significant bit) gets propagated to its parent. After $m +$ log n steps, the up sweep is completed. Using a similar method, the down sweep is also calculated in $m +$ log n steps. The total number of steps is therefore $2(m +$ log $n)$. The down sweep can actually start as soon as the first bit of the up sweep reaches the top, reducing the number of steps to $m + 2$ log n.

B. Hardware Implementation of Tree Scan

We now discuss in more detail the hardware needed to implement the bit-pipelined tree scan for the two primitive scan operations +-**scan** and **max-scan**. Fig. 14 shows an implementation of a unit of the binary tree. Each unit consist of two identical state machines, a variable-length shift register and a one bit register. The control for a unit consists of a clock, a clear signal, and an operation specification, which specifies whether to execute a +-**scan** or a **max-scan**. The control signals are identical on all units. The units are connected in a balanced binary tree, as shown in Fig. 13, with two single bit unidirectional wires along every edge.

The *shift register* acts as a first-in-first-out buffer (FIFO), with bits entered on one end and removed from the other. One bit is shifted on each clock signal. The length of the register depends on the depth of the unit in the tree. A unit at level i from the top needs a register of length $2i$ bits. The maximum length is therefore 2 log n bits. The length of the shift register can either be hardwired into each unit, in which case different levels of the tree would require different units, or could be

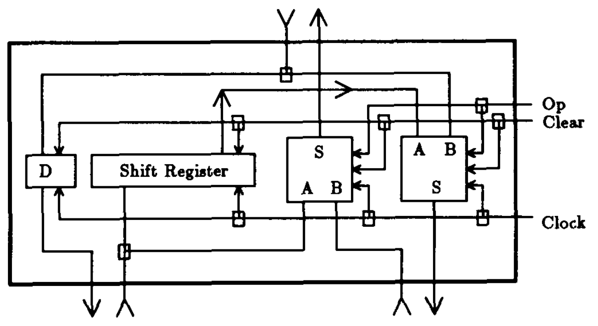
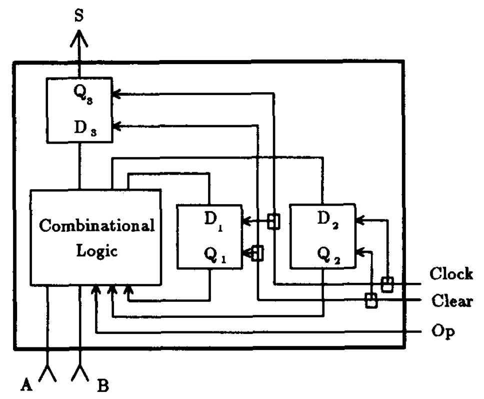


Fig. 14. Diagram of a unit needed to implement the tree algorithm. It consists of a shift register (which acts as a first-in-first-out buffer), a one bit register (a D type flip-flop), and two identical sum state machines. These units are arranged in a tree as shown in Fig. 13. Each wire is a single bit wire.



$$\begin{array}{rcccl} & & \underline{\text{For max-scan}} & & \underline{\text{For +-scan}} \\ S & = & \mathrm{Op}(B\overline{Q_1}+A\overline{Q_2}) & + & \overline{\mathrm{Op}}(A\oplus B\oplus Q_1) \\ D_1 & = & \mathrm{Op}(Q_1+\overline{B}A\overline{Q_2}) & + & \overline{\mathrm{Op}}(AB+AQ_1+BQ_1) \\ D_2 & = & \mathrm{Op}(Q_2+\overline{A}B\overline{Q_1}) & & \end{array}$$

Fig. 15. Diagram of the sum state machine. It consists of three d-type flip-flops and some combinational logic. If the signal *Op* is true, the circuit executes a **max-scan**. If the signal *Op* is false, the circuit executes a **+-scan**. In the logic equations, the symbol $\oplus$ is an Exclusive OR, and the symbol + is an Inclusive OR. This state machine fits into a unit as shown in Fig. 14.

controlled by some logic, in which case all units could be identical, but the level number would need to be stored on each unit.

The *sum state machine* consists of three bits of state and a five-input, three-output combinational logic circuit. Fig. 15 shows its basic layout and the required logic. The *Op* control specifies whether to execute a **+-scan** or a **max-scan**. Two bits of state are needed for the **max-scan** to keep track of whether A is greater, equal, or lesser than B (if Q_1 is set, A is greater, if Q_2 is set, B is greater). The **+-scan** only uses one bit of state to store the carry flag (Q_1). The third bit of state is used to hold the output value (S) for one clock cycle.

To execute a scan on a tree of such units, we reset the state machines in all units with the *clear* signal and set the *Op* signal to execute either a **+-scan** or **max-scan**. We must tie the parent input of the root unit low (0). We then simply insert one bit of the operand at the leaves on each clock cycle. In a **max-scan**, the bits are inserted starting at the most significant bit, and in a **+-scan** the bits are inserted starting at the least significant bit. After 2 log n steps, the result will start returning at the leaves one bit on each clock cycle. We do not even need to change anything when going from the up sweep to the down sweep: when the values reach the root, they are automatically reflected back down since the shift register at the root has length 0. The total hardware needed for scanning n values is $n - 1$ shift registers and $2(n - 1)$ sum state machines. The units are simple so it should be easy to place many on a chip.

Perhaps more importantly than the simplicity of each unit is the fact that the units are organized in a tree. The tree organization has two important practical properties. First, only two wires are needed to leave every branch of the tree. So, for example, if there are several processors per chip, only a pair of wires are needed to leave that chip, and if there are many processors on a board, only a pair of wires are needed to leave the board. Second, a tree circuit is much easier to synchronize than other structures such as grids, hypercubes, or butterfly networks. This is because the same tree used for the scans can be used for clock distribution. Such a clock distribution gets rid of the clock skew problem[8] and makes it relatively easy to run the circuit extremely fast.

C. An Example System

We now consider an example system to show how the scan circuit might be applied in practice. We consider a 4096 processor parallel computer with 64 processors on each board and 64 boards per machine. To implement the scan primitives on such a machine, we could use a single chip on each board that has 64 inputs and 1 output and acts as six levels of the tree. Such a chip would require 126 sum state machines and 63 shift registers—such a chip is quite easy to build with today's technology. We could use one more of these chips to combine the pair of wires from each of the 64 boards.

If the clock period is 100 ns, a scan on a 32 bit field would require 5 μs. This time is considerably faster than the routing time of existing parallel computers such as the BBN Butterfly or the Thinking Machines Connection Machine. With a more aggressive clock such as the 10 ns clock being looked at by BBN for the Monarch[9] [2], this time would be reduced to 0.5 μs—twice as fast as the best case global access time expected on the Monarch.

In most existing and proposed tightly connected parallel computers [22], [36], [2], [41], the cost of the communication network is between 1/3 and 1/2 the cost of the computer. It is unlikely that the suggested scan network will be more than 1 percent of the cost of a computer.

D. Simulating All Scans with a +-Scan and Max-Scan

All the scans discussed in this paper, including the segmented versions, can be implemented with just two scans: integer versions of the **+-scan** and **max-scan**. This sec-

[8] When there are many synchronous elements in a system, the small propagation time differences in different paths when distributing the clock signals can cause significant clock time differences at the elements.

[9] Because of the tree structure, it would actually be much easier to run a clock at 10 ns on a scan network than it is for the communication network of the Monarch.

```
define seg-max-scan(A, SFlag){
        Seg-Number ← SFlag + enumerate(SFlag);
        B ← append(Seg-Num, A);
        C ← extract-bot(max-scan(A'));
    if SFlag then C else 0}
```

A	=	[5	1	3	4	3	9	2	6]
SFlag	=	[T	F	T	F	F	F	T	F]
Seg-Number	=	[1	1	2	2	2	2	3	3]
B	=	[1,5	1,1	2,3	2,4	2,3	2,9	3,2	3,6]
C	=	[0	5	5	3	4	4	9	2]
Result	=	[0	5	0	3	4	4	0	2]

Fig. 16. The implementation of a segmented **max-scan**.

tion discusses the implementation. The implementation requires access to the bit representation of the numbers.

An integer **min-scan** can be implemented by inverting the source, executing a **max-scan**, and inverting the result. A floating-point **max-scan** and **min-scan** can be implemented by flipping the exponent and significant if the sign bit is set, executing the signed version, and flipping the exponent and significant of the result back based on the sign bit. The **or-scan** and **and-scan** can be implemented with a 1-bit **max-scan** and **min-scan**, respectively. The implementation of the floating-point +-**scan** is described elsewhere [7].

A segmented **max-scan** is implemented by first enumerating the segment bits, appending the result (plus 1 in positions where the flag is set) to the original numbers, executing an unsegmented **max-scan**, and removing the appended bits (see Fig. 16). A segmented +-**scan** is implemented by executing an unsegmented +-**scan**, copying the first element in each segment across the segment using a segmented **copy** (can be implemented with a segmented **max-scan**), and subtracting this element.

The backward scans can be implemented by simply reading the vector into the processors in reverse order.

IV. Conclusions

This paper described a study of the effects of adding two scan primitives as unit-time primitives to the PRAM models. The study shows that these scan primitives take no longer to execute than parallel memory references, both in practice and in theory, but yet can improve the asymptotic running time, and the description of many algorithms. We wonder whether there might be other primitives that can be cheaply implemented on a parallel architecture. One such primitive might be a merge primitive that merges two sorted vectors. As shown by Batcher [4], this can be executed in a single pass of an omega network.

We hope that this paper will prompt researchers to question the validity of the pure PRAM models when designing and analyzing algorithms—especially when it comes down to trying to remove $O(\log \log n)$ factors off of the running time of an algorithm.

Appendix
A Short History of the Scan Operations

This appendix gives a brief history of the scan operations. Scans were suggested by Iverson in the mid 1950's as operations for the language APL [25]. A parallel circuit to execute the operation was first suggested by Ofman in the early 1960's [35] to be used to add binary numbers—the following routine executes addition on two binary numbers with their bits spread across two vectors A and B ($\otimes$ means Exclusive or):

$$(A \otimes B) \otimes \textbf{seg-or-scan}(AB, \overline{AB}).$$

A general scan operator was suggested by Iverson in the mid 1960's for the language APL. The history of the scan operator in APL is actually quite complex. It did not appear in the original definition [25], but appears in some but not all subsequent definitions. A parallel implementation of scans on a perfect shuffle network was later suggested by Stone [44] to be used for polynomial evaluation—the following routine evaluates a polynomial with a vector of coefficients A and variable x at the head of another vector X:

$$A \times \times\textbf{-scan}(\textbf{copy}(X)).$$

Ladner and Fisher first showed an efficient general-purpose circuit for implementing the scan operations [28]. Wyllie first showed how the scan operation can be executed on a linked list using the PRAM model [48]. Brent and Kung, in the context of binary addition, first showed an efficient VLSI layout for a scan circuit [10]. Schwartz [40] and, independently, Mago [32] first suggested the segmented versions of the scans. More recent work on implementing scan operations in parallel include the work of Fich [15], which demonstrates a more efficient implementation of the scan operations, and of Lubachevsky and Greenberg [31], which demonstrates the implementation of the scan operation on asynchronous machines.

As concerns terminology, *scan* is the original name given to the operation. Ladner and Fisher introduced the term *parallel prefix operation*. Schwartz used the term *all partial sums*. I find the original term, scan, more concise and more flexible—it, for example, can be used as a verb, as in "the algorithm then scans the vector."

Acknowledgment

Many thanks to C. Leiserson for his ideas, support, and enthusiasm, A. Ranade for first pointing out to me that scans can be bit pipelined, and to A. Goldberg, D. Hillis, T. Knight, J. Rose, and G. Steele for their helpful comments.

References

[1] M. Ajtai, J. Komlos, and E. Szemeredi, "An $O(n \log n)$ sorting network," in *Proc. ACM Symp. Theory Comput.*, Apr. 1983, pp. 1-9.
[2] D. C. Allen, "The BBN multiprocessors: Butterfly and Monarch," in *Proc. Princeton Conf. Supercomput. Stellar Dynamics*, June 1986.
[3] B. Awerbuch and Y. Shiloach, "New connectivity and MSF algorithms for Ultracomputer and PRAM," in *Proc. ACM Symp Theory Comput.*, 1985, pp. 175-179.
[4] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, 1968, pp. 307-314.
[5] C. Berge and A. Ghouila-Houri, *Programming, Games, and Transportation Networks.* New York: Wiley, 1965.
[6] H. J. Berliner, "A chronology of computer chess and its literature," *Artif. Intell.*, vol. 10, no. 2, 1978.
[7] G. E. Blelloch, "Scan primitives and parallel vector models," Ph.D. dissertation, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, Nov. 1988.
[8] G. E. Blelloch and J. J. Little, "Parallel solutions to geometric problems on the scan model of computation," in *Proc. Int. Conf. Parallel Processing*, vol. 3, Aug. 1988, pp. 218-222.

[9] O. Borůvka, "O jistém problén minimálim," *Práca Moravské Přírodovědecké Společnosti*, vol. 3, pp. 37-58, 1926 (In Czech.).
[10] R. P. Brent and H. T. Kung, "The chip complexity of binary arithmetic," in *Proc. ACM Symp. Theory Comput.*, 1980, pp. 190-200.
[11] R. Cole, "Parallel merge sort," in *Proc. Symp. Foundations Comput. Sci.*, Oct. 1986, pp. 511-516.
[12] R. Cole and U. Vishkin, "Approximate scheduling, exact scheduling, and applications to parallel algorithms," in *Proc. Symp. Foundations Comput. Sci.*, 1986, pp. 478-491.
[13] ——, "Faster optimal parallel prefix sums and list ranking," Tech. Rep. Ultracomputer Note 117, New York Univ., Feb. 1987.
[14] C. C. Elgot and A. Robinson, "Random access stored program machines," *J. ACM*, vol. no. 4, pp. 365-399, 1964.
[15] F. E. Fich, "New bounds for parallel prefix circuits," in *Proc. ACM Symp. Theory Comput.*, Apr. 1983, pp. 100-109.
[16] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proc. ACM Symp. Theory Comput.*, 1978, pp. 114-118.
[17] F. Gavril, "Merging with parallel processors," *Commun. ACM*, vol. 18, no. 10, pp. 588-591, 1975.
[18] H. Gazit, G. L. Miller, and S.-H. Teng, *Optimal Tree Contraction in the EREW Model*. New York: Plenum, 1988, pp. 139-156.
[19] L. M. Goldschlager, "A universal interconnection pattern for parallel computers," *J. ACM*, vol. 29, no. 3, pp. 1073-1086, 1982.
[20] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing a MIMD, shared-memory parallel machine," *IEEE Trans. Comput.*, vol. C-32, pp. 175-189, 1983.
[21] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors," *ACM Trans. Programming Languages Syst.*, vol. 5, no. 2, Apr. 1983.
[22] W. Daniel Hillis, *The Connection Machine*. Cambridge, MA: MIT Press, 1985.
[23] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, no. 8, pp. 461-464, 1979.
[24] C. A. R. Hoare, "Quicksort," *Comput. J.*, vol. 5, no. 1, pp. 10-15, 1962.
[25] K. E. Iverson, *A Programming Language*. New York: Wiley, 1962.
[26] D. E. Knuth, *Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
[27] C. P. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," in *Proc. Int. Conf. Parallel Processing*, Aug. 1985, pp. 180-185.
[28] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831-838, Oct. 1980.
[29] F. T. Leighton, "Tight bounds on the complexity of parallel sorting," in *Proc. ACM Symp. Theory Comput.*, May 1984, pp. 71-80.
[30] C. E. Leiserson, "Area-efficient layouts (for VLSI)," in *Proc. Symp. Foundations Comput. Sci.*, 1980.
[31] B. D. Lubachevsky and A. G. Greenberg, "Simple, efficient asynchronous parallel prefix algorithms," in *Proc. Int. Conf. Parallel Processing*, Aug. 1987, pp. 66-69.
[32] G. A. Mago, "A network of computers to execute reduction languages," *Int. J. Comput. Inform. Sci.*, 1979.
[33] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," in *Proc. Symp. Foundations Comput. Sci.*, Oct. 1985, pp. 478-489.
[34] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*. New York: McGraw-Hill, 1979.
[35] Y. Ofman, "On the algorithmic complexity of discrete functions," *Cybern. Contr. Theory, Sov. Phys Dokl.*, vol. 7, no. 7, pp. 589-591, Jan. 1963.
[36] G. F. Pfister and V. A. Norton, "'Hot spot' contention and combining in multistage interconnection networks," in *Proc. Int. Conf. Parallel Processing*, Aug. 1985, pp. 790-797.
[37] A. G. Ranade, "Fluent parallel computation," Ph.D. dissertation, Yale Univ., Dep. Comput. Sci., New Haven, CT, 1989.
[38] J. B. Salem, "*Render: A data parallel approach to polygon rendering," Tech. Rep. VZ88-2, Thinking Machines Corp., Jan. 1988.
[39] C. Savage and J. Ja'Ja', "Fast, efficient parallel algorithms for some graph problems," *SIAM*, vol. 10, no. 4, pp. 683-691, 1981.
[40] J. T. Schwartz, "Ultracomputers," *ACM Trans. Programming Languages Syst.*, vol. 2, no. 4, pp. 484-521, Oct. 1980.
[41] C. L. Seitz, "The Cosmic Cube," *Commun. ACM*, vol. 28, no. 1, pp. 22-33, Jan. 1985.
[42] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," *J. Algorithms*, vol. 2, no. 1, pp. 88-102, 1981.
[43] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *J. Algorithms*, vol. 3, pp. 57-67, 1982.
[44] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, no. 2, pp. 53-161, 1971.
[45] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA: SIAM, 1983.
[46] Thinking Machines Corp., "Connection Machine parallel instruction set (PARIS)," July 1986.
[47] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Commun. ACM*, vol. 20, pp. 263-271, 1977.
[48] J. C. Wyllie, "The complexity of parallel computations," Tech. Rep. TR-79-387, Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Aug. 1979.

Guy E. Blelloch received the B.A. degree in physics in 1983 from Swarthmore College, Swarthmore, PA, and the M.S. and Ph.D. degree in computer science in 1986 and 1988, respectively, from the Massachusetts Institute of Technology, Cambridge.

He is an Assistant Professor at Carnegie Mellon University, Pittsburgh, PA. His research interests include data-parallel algorithms, languages, and compilers. He has worked or consulted for Thinking Machines Corporation since 1985.