Prefix Sums and Their Applications

Guy E. Blelloch
November 1990
CMU-CS-90-1903

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report will appear as a chapter in the book "Synthesis of Parallel Algorithms", Editor: John H. Reif.

Abstract

Experienced algorithm designers rely heavily on a set of building blocks and on the tools needed to put the blocks together into an algorithm. The understanding of these basic blocks and tools is therefore critical to the understanding of algorithms. Many of the blocks and tools needed for parallel algorithms extend from sequential algorithms, such as dynamic-programming and divide-and-conquer, but others are new. This paper introduces one of the simplest and most useful building blocks for parallel algorithms: the *all-prefix-sums* operation. The paper defines the operation, shows how to implement it on a P-RAM and illustrates many applications of the operation. In addition to being a useful building block, the all-prefix-sums operation is a good example of a computation that seems inherently sequential, but for which there is an efficient parallel algorithm.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

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 the U.S. government.

Keywords: Scan operation, parallel algorithms, all-prefix-sums, linear-recurrence, parallel quicksort

1 Introduction[1]

Experienced algorithm designers rely heavily on a set of building blocks and on the tools needed to put the blocks together into an algorithm. The understanding of these basic blocks and tools is therefore critical to the understanding of algorithms. Many of the blocks and tools needed for parallel algorithms extend from sequential algorithms, such as dynamic-programming and divide-and-conquer, but others are new.

This paper introduces one of the simplest and most useful building blocks for parallel algorithms: the *all-prefix-sums* operation. The paper defines the operation, shows how to implement it on a P-RAM and illustrates many applications of the operation. In addition to being a useful building block, the all-prefix-sums operation is a good example of a computation that seems inherently sequential, but for which there is an efficient parallel algorithm. The operation is defined as follows:

Definition: *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$, *and an ordered set of* n *elements*

$$[a_0, a_1, ..., a_{n-1}],$$

and returns the ordered set

$$[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})].$$

For example, if $\oplus$ is addition, then the all-prefix-sums operation on the ordered set

$$[3 \quad 1 \quad 7 \quad 0 \quad 4 \quad 1 \quad 6 \quad 3],$$

would return

$$[3 \quad 4 \quad 11 \quad 11 \quad 14 \quad 16 \quad 22 \quad 25].$$

The uses of the all-prefix-sums operation are extensive. Here is a list of some of them:

1. To lexically compare strings of characters. For example, to determine that "`strategy`" should appear before "`stratification`" in a dictionary (see Problem 2).
2. To add multi precision numbers. These are numbers that cannot be represented in a single machine word (see Problem 3).
3. To evaluate polynomials (see Problem 6).
4. To solve recurrences. For example, to solve the recurrences $x_i = a_i x_{i-1} + b_i x_{i-2}$ and $x_i = a_i + b_i / x_{i-1}$ (see Section 4).
5. To implement radix sort (see Section 3).

[1]This paper was originally written as a chapter to a book so all the citations appear at the end.

```
proc all-prefix-sums-Array(Out, In)
  i ← 0
  sum ← In[0]
  Out[0] ← sum
  while (i < length)
    i ← i + 1
    sum ← sum + In[i]
    Out[i] ← sum
```

```
proc all-prefix-sums-List(Out, In)
  i ← 0
  sum ← In[0].value
  Out[0] ← sum
  while (In[i].pointer ≠ End-of-List)
    i ← In[i].pointer
    sum ← sum + In[i].value
    Out[i] ← sum
```

Figure 1: Sequential algorithms for calculating the all-prefix-sums operation with operator + on an array and on a linked-list. In the list version, each element of In consists of two fields: a value (.value), and a pointer to the next position in the list (.pointer).

6. To implement quicksort (see Section 5.1).
7. To solve tridiagonal linear systems (see Problem 12).
8. To delete marked elements from an array (see Section 3).
9. To dynamically allocate processors (see Section 6).
10. To perform lexical analysis. For example, to parse a program into tokens.
11. To search for regular expressions. For example, to implement the UNIX grep program.
12. To implement some tree operations. For example, to find the depth of every vertex in a tree.
13. To label components in two dimensional images.

In fact, all-prefix-sums operations using addition, minimum and maximum are so useful in practice that they have been included as primitive instructions in some machines. Researchers have also suggested that a subclass of the all-prefix-sums operation be added to the P-RAM model as a "unit time" primitive since this class theoretically can run as fast as a memory access of the shared memory in the P-RAM.

Before describing the implementation we must consider how the definition of the all-prefix-sums operation relates to the P-RAM model. The definition states that the operation takes an ordered set, but does not specify how the ordered set is laid out in memory. One way to lay out the elements is in contiguous locations of an array; another way is to use a linked-list with pointers from each element to the next. It turns out that both forms of the operation have uses. In the examples listed above, the component labeling and some of the tree operations require the linked-list version, while the other examples can use the array version.

Sequentially, both versions are easy to compute (see Figure 1). The array version steps down the array, adding each element into a sum and writing the sum back, while the linked-list version follows the pointers while keeping the running sum and writing it back. The algorithms in Figure 1 for both versions are inherently sequential: to calculate a value at any step, the result of the previous step is needed. The algorithms therefore require $O(n)$ time. To execute the all-prefix-sums operation in parallel, the algorithms must be changed significantly.

The remainder of this paper is just concerned with the array all-prefix-sums operation. We will henceforth use the term *scan* for this operation.[2]

Definition: *The* scan *operation is an array all-prefix-sums operation.*

Sometimes it is useful for each element of the result array to contain the sum of all the previous elements, but not the element itself. We call such an operation, a *prescan.*

Definition: *The* prescan *operation takes a binary associative operator* $\oplus$ *with identity* I, *and an array of* n *elements*

$$[a_0, a_1, ..., a_{n-1}],$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})].$$

A prescan can be generated from a scan by shifting the array right by one and inserting the identity. Similarly, the scan can be generated from the prescan by shifting left, and inserting at the end the sum of the last element of the prescan and the last element of the original array.

2 Implementation

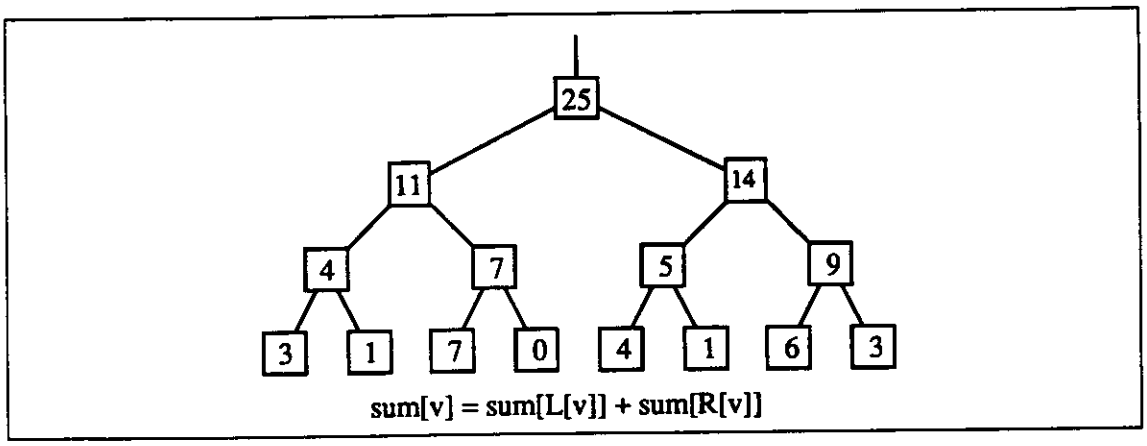
This section describes an algorithm for executing the scan operation. For p processors and an array of length n on an EREW P-RAM, the algorithm has a time complexity of $O(n/p + \lg p)$. The algorithm is simple and well suited for direct implementation in hardware.

Before describing the scan operation, we consider a simpler problem, that of generating only the final element of the scan. We call this the *reduce* operation.

Definition: *The* reduce *operation takes a binary associative operator* $\oplus$ *with identity* i, *and an ordered set* $[a_0, a_1, ..., a_{n-1}]$ *of* n *elements, and returns the value* $a_0 \oplus a_1 \oplus ... \oplus a_{n-1}$.

Again we consider only the case where the ordered set is kept in an array. A balanced binary tree can be used to implement the reduce operation by laying the tree over the values, and using $\oplus$ to sum pairs at each vertex (see Figure 2a). The correctness of the result relies

[2]The term scan comes from the computer language APL.



(a) Executing a +-reduce on a tree.

for d **from** 0 **to** $(\lg n) - 1$
  in parallel for i **from** 0 **to** $n - 1$ **by** 2^{d+1}
    $a[i + 2^{d+1} - 1] \leftarrow a[i + 2^d - 1] + a[i + 2^{d+1} - 1]$

Step	Array in Memory							
0	[**3**	**1**	**7**	**0**	**4**	**1**	**6**	**3**]
1	[3	**4**	7	**7**	4	**5**	6	**9**]
2	[3	4	7	**11**	4	5	6	**14**]
3	[3	4	7	11	4	5	6	**25**]

(b) Executing a +-reduce on a P-RAM.

Figure 2: An example of the reduce operation when $\oplus$ is integer addition. The boxes in (b) show the locations that are modified on each step. The length of the array is n and must be a power of two. The final result will reside in $a[n-1]$.

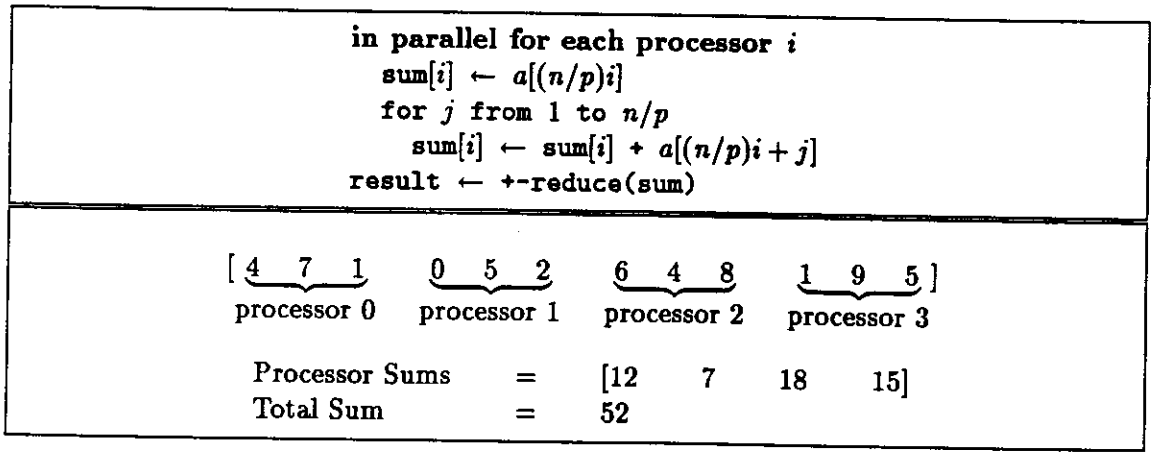


Figure 3: The +-reduce operation with more elements than processors. We assume that n/p is an integer.

on $\oplus$ being associative. The operator, however, does not need to be commutative since the order of the operands is maintained. On an EREW P-RAM, each level of the tree can be executed in parallel, so the implementation can step from the leaves to the root of the tree (see Figure 2b); we call this an up-sweep. Since the tree is of depth $\lceil \lg n \rceil$, and one processor is needed for every pair of elements, the algorithm requires $O(\lg n)$ time and $n/2$ processors.

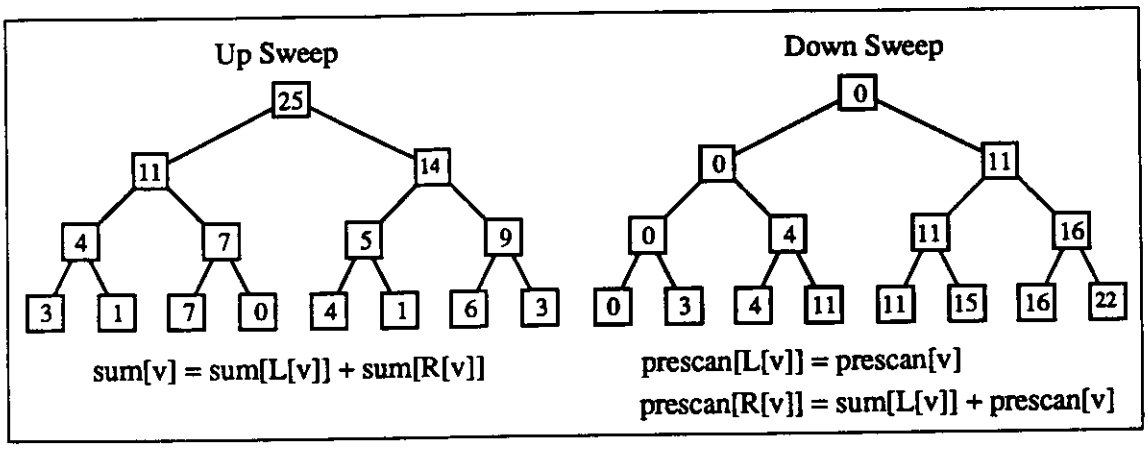
If we assume a fixed number of processors p, with $n > p$, then each processor can sum an n/p section of the array to generate a processor sum; the tree technique can then be used to reduce the processor sums (see Figure 3). The time taken to generate the processor sums is $\lceil n/p \rceil$, so the total time required on an EREW P-RAM is:

$$T_R(n,p) = \lceil n/p \rceil + \lceil \lg p \rceil = O(n/p + \lg p). \tag{1}$$

When $n/p \geq \lg p$ the complexity is $O(n/p)$. This time is an optimal speedup over the sequential algorithm given in Figure 1.

We now return to the scan operation. We actually show how to implement the prescan operation; the scan is then determined by shifting the result and putting the sum at the end. If we look at the tree generated by the reduce operation, it contains many partial sums over regions of the array. It turns out that these partial sums can be used to generate all the prefix sums. This requires executing another sweep of the tree with one step per level, but this time starting at the root and going to the leaves (a down-sweep). Initially, the identity element is inserted at the root of the tree. On each step, each vertex at the current level passes to its left child its own value, and it passes to its right son, $\oplus$ applied to the value from the left child from the up-sweep and its own value (see Figure 4a).

Let us consider why the down-sweep works. We say that vertex x *precedes* vertex y if x appears before y in the preorder traversal of the tree (depth first, from left to right).



(a) Executing a +-prescan on a tree.

```
procedure down-sweep(A)
  a[n − 1] ← 0
  for d from (lg n) − 1 downto 0
    in parallel for i from 0 to n − 1 by 2^(d+1)
      t ← a[i + 2^d − 1]                          % Save in temporary
      a[i + 2^d − 1] ← a[i + 2^(d+1) − 1]         % Set left child
      a[i + 2^(d+1) − 1] ← t + a[i + 2^(d+1) − 1] % Set right child
```

	Step	Array in Memory							
	0	[**3**	**1**	**7**	**0**	**4**	**1**	**6**	**3**]
up	1	[3	**4**	7	**7**	4	**5**	6	**9**]
	2	[3	4	7	**11**	4	5	6	**14**]
	3	[3	4	7	11	4	5	6	**25**]
clear	4	[3	4	7	11	4	5	6	**0**]
down	5	[3	4	7	**0**	4	5	6	**11**]
	6	[3	**0**	7	**4**	4	**11**	6	**16**]
	7	[**0**	**3**	**4**	**11**	**11**	**15**	**16**	**22**]

(b) Executing a +-prescan on a P-RAM.

Figure 4: A parallel prescan on a tree using integer addition as the associative operator $\oplus$.

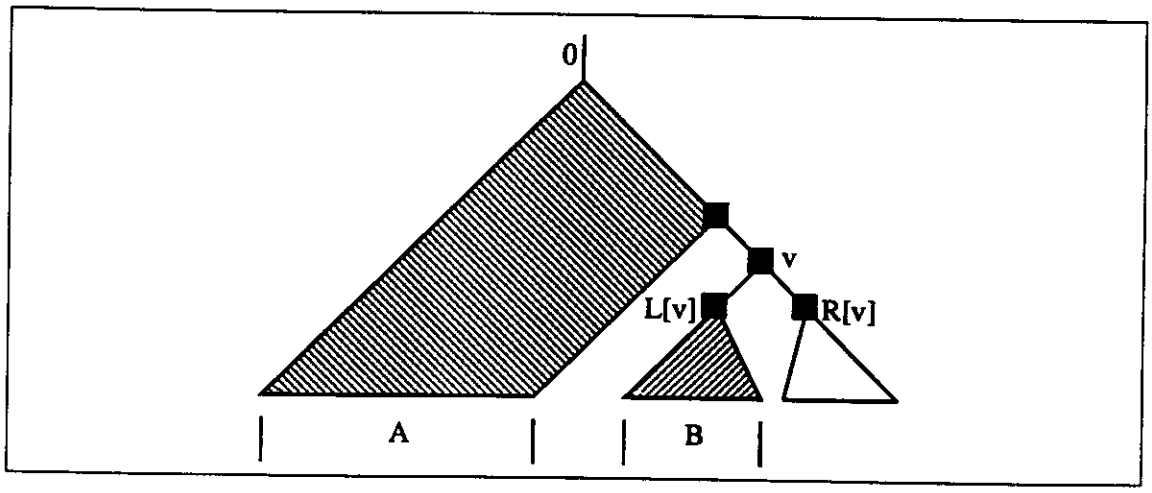


Figure 5: Illustration for Theorem 1.

Theorem 1 *After a complete down-sweep, each vertex of the tree contains the sum of all the leaf values that precede it.*

Proof: The proof is inductive from the root: we show that if a parent has the correct sum, both children must have the correct sum. The root has no elements preceding it, so its value is correctly the identity element.

Consider Figure 5. The left child of any vertex has exactly the same leaves preceding it as the vertex itself (the leaves in region A in the figure). This is because the preorder traversal always visits the left child of a vertex immediately after the vertex. By the induction hypothesis, the parent has the correct sum, so it need only copy this sum to the left child.

The right child of any vertex has two sets of leaves preceding it, the leaves preceding the parent (region A), and the leaves at or below the left child (region B). Therefore, by adding the parent's down-sweep value, which is correct by the induction hypothesis, and the left-child's up-sweep value, the right-child will contain the sum of all the leaves preceding it. □

Since the leaf values that precede any leaf are the values to the left of it in the scan order, the values at the leaves are the results of a left-to-right prescan. To implement the prescan on an EREW P-RAM, the partial sums at each vertex must be kept during the up-sweep so they can be used during the down-sweep. We must therefore be careful not to overwrite them. In fact, this was the motivation for putting the sums on the right during the reduce in Figure 2b. Figure 4b shows the P-RAM code for the down-sweep. Each step can execute in parallel, so the running time is $2\lceil \lg n \rceil$.

If we assume a fixed number of processors p, with $n > p$, we can use a similar method to that in the reduce operation to generate an optimal algorithm. Each processor first sums an n/p section of the array to generate a processor sum, the tree technique is then used to prescan the processor sums. The results of the prescan of the processor sums are used

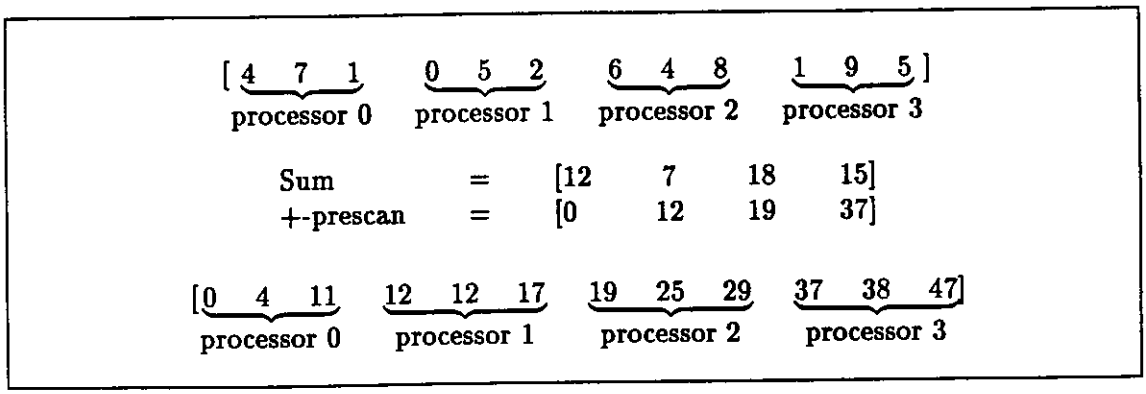


Figure 6: A +-prescan with more elements than processors.

as an offset for each processor to prescan within its n/p section (see Figure 6). The time complexity of the algorithm is:

$$T_S(n,p) = 2(\lceil n/p \rceil + \lceil \lg p \rceil) = O(n/p + \lg n) \tag{2}$$

which is the same order as the reduce operation and is also an optimal speedup over the sequential version when $n/p \geq \lg p$.

This section described how to implement the scan (prescan) operation. The rest of the paper discusses its applications.

3 Line-of-Sight and Radix-Sort

As an example of the use of a scan operation, consider a simple line-of-sight problem. The line-of-sight problem is: given a terrain map in the form of a grid of altitudes and an observation point X on the grid, find which points are visible along a ray originating at the observation point (see Figure 7).

A point on a ray is visible if and only if no other point between it and the observation point has a greater vertical angle. To find if any previous point has a greater angle, the altitude of each point along the ray is placed in a vector (the *altitude vector*). These altitudes are then converted to angles and placed in the *angle vector* (see Figure 7). A prescan using the operator maximum (max-prescan) is then executed on the *angle vector*, which returns to each point the maximum previous angle. To test for visibility each point only needs to compare its angle to the result of the max-prescan. This can be generalized to finding all visible points on the grid. For n points on a ray, the complexity of the algorithm is the complexity of the scan, $T_S(n,p) = O(n/p + \lg n)$ on an EREW P-RAM.

We now consider another example, a radix sort algorithm. The algorithm loops over the bits of the keys, starting at the lowest bit, executing a split operation on each iteration (assume all keys have the same number of bits). 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

```
procedure line-of-sight(altitude)
  in parallel for each index i
    angle[i] ← arctan(scale × (altitude[i] - altitude[0])/ i)
  max-previous-angle ← max-prescan(angle)
  in parallel for each index i
    if (angle[i] > max-previous-angle[i])
       result[i] ← "visible"
    else
       result[i] ← not "visible"
```

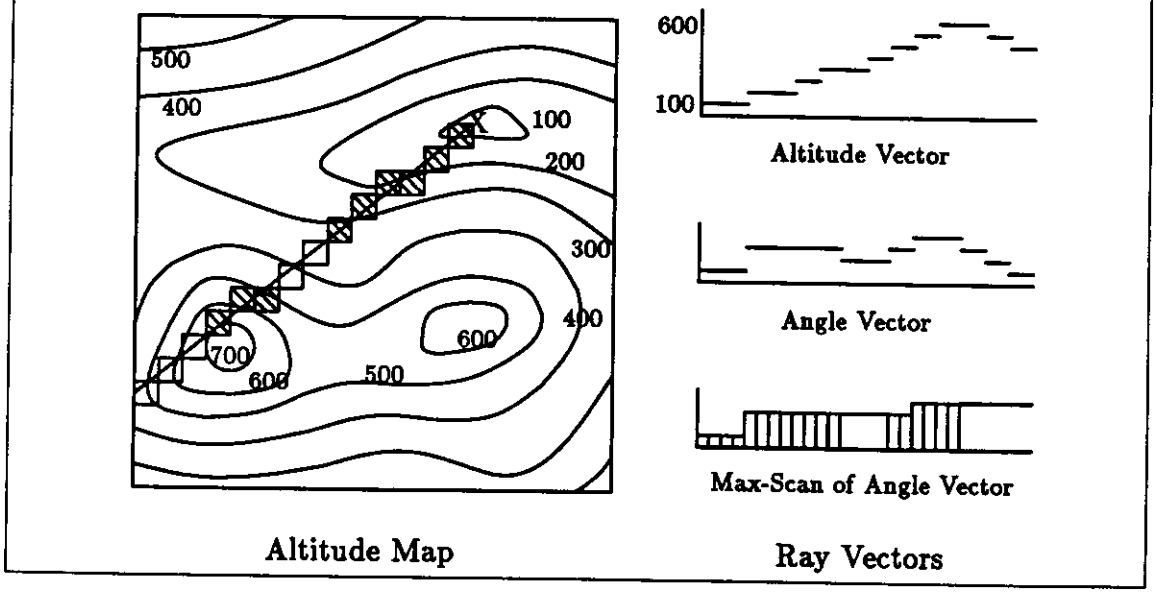


Figure 7: The line-of-sight algorithm for a single ray. The X marks the observation point. The visible points are shaded. A point on the ray is visible if no previous point has a greater angle.

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

Figure 8: An example of the split radix sort on a vector containing three bit values. The A⟨n⟩ notation signifies extracting the n^{th} bit of each element of the vector A. The split operation packs elements with a 0 flag to the bottom and with a 1 flag to the top.

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. Figure 8 shows an example of the sort.

We now consider how the split operation can be implemented using a scan. The basic idea is to determine a new index for each element and then permute the elements to these new indices using an exclusive write. To determine the new indices for elements with a 0 in the bit, we invert the flags and execute a prescan with integer addition. To determine the new indices of elements with a 1 in the bit, we execute a +-scan in reverse order (starting at the top of the vector) and subtract the results from the length of the vector n. Figure 9 shows an example of the split operation along with code to implement it.

Since the split operation just requires two scan operations, a few steps of exclusive memory accesses, and a few parallel arithmetic operations, it has the same asymptotic complexity as the scan: $O(n/p + \lg p)$ on an EREW P-RAM.[3] If we assume that n keys are each $O(\lg n)$ bits long, then the overall algorithm runs in time:

$$O((\frac{n}{p} + \lg p)\lg n) = O(\frac{n}{p}\lg n + \lg n \lg p).$$

4 Recurrence Equations

This section shows how various recurrence equations can be solved using the scan operation. A recurrence is a set of equations of the form

$$x_i = f_i(x_{i-1}, x_{i-2}, \cdots, x_{i-m}), \qquad m \le i < n \tag{3}$$

[3]On an CREW P-RAM we can use the scan described in [6] to get a time of $O(n/p + \lg p/\lg\lg p)$.

```
procedure split(A, Flags)
  I-down ← +-prescan(not(Flags))
  I-up   ← n - +-scan(reverse-order(Flags))
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)
```

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
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]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]

Figure 9: 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. The permute writes each element of A to the index specified by the corresponding position in Index.

along with a set of initial values $x_0, \cdots, x_{m-1}$.

The scan operation is the special case of a recurrence of the form

$$x_i = \begin{cases} a_0 & i = 0 \\ x_{i-1} \oplus a_i & 0 < i < n, \end{cases} \tag{4}$$

where $\oplus$ is any binary associative operator. This section shows how to reduce a more general class of recurrences to equation (4), and therefore how to use the scan algorithm discussed in Section 2 to solve these recurrences in parallel.

4.1 First-Order Recurrences

We initially consider *first-order* recurrences of the following form

$$x_i = \begin{cases} b_0 & i = 0 \\ (x_{i-1} \otimes a_i) \oplus b_i & 0 < i < n, \end{cases} \tag{5}$$

where the a_i's and b_i's are sets of n arbitrary constants (not necessarily scalars) and $\oplus$ and $\otimes$ are arbitrary binary operators that satisfy three restrictions:

1. $\oplus$ is associative (i.e. $(a \oplus b) \oplus c = a \oplus (b \oplus c)$).
2. $\otimes$ is semiassociative (i.e. there exists a binary associative operator $\odot$ such that $(a \otimes b) \otimes c = a \otimes (b \odot c)$).
3. $\otimes$ distributes over $\oplus$ (i.e. $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$).

The operator $\odot$ is called the *companion operator* of $\otimes$. If $\otimes$ is fully associative, then $\odot$ and $\otimes$ are equivalent.

We now show how (5) can be reduced to (4). Consider the set of pairs

$$c_i = [a_i, b_i] \tag{6}$$

and define a new binary operator $\bullet$ as follows:

$$c_i \bullet c_j \equiv [c_{i,a} \odot c_{j,a},\ (c_{i,b} \otimes c_{j,a}) \oplus c_{j,b}] \tag{7}$$

where $c_{i,a}$ and $c_{i,b}$ are the first and second elements of c_i, respectively.

Given the conditions on the operators $\oplus$ and $\otimes$, the operator $\bullet$ is associative as we show below:

$$\begin{aligned}
(c_i \bullet c_j) \bullet c_k &= [c_{i,a} \odot c_{j,a},\ (c_{i,b} \otimes c_{j,a}) \oplus c_{j,b}] \bullet c_k \\
&= [(c_{i,a} \odot c_{j,a}) \odot c_{k,a},\ (((c_{i,b} \otimes c_{j,a}) \oplus c_{j,b}) \otimes c_{k,a}) \oplus c_{k,b}] \\
&= [c_{i,a} \odot (c_{j,a} \odot c_{k,a}),\ ((c_{i,b} \otimes c_{j,a}) \otimes c_{k,a}) \oplus ((c_{j,b} \otimes c_{k,a}) \oplus c_{k,b})] \\
&= [c_{i,a} \odot (c_{j,a} \odot c_{k,a}),\ (c_{i,b} \otimes (c_{j,a} \odot c_{k,a})) \oplus ((c_{j,b} \otimes c_{k,a}) \oplus c_{k,b})] \\
&= c_i \bullet [c_{j,a} \odot c_{k,a},\ (c_{j,b} \otimes c_{k,a}) \oplus c_{k,b}] \\
&= c_i \bullet (c_j \bullet c_k)
\end{aligned}$$

We now define the ordered set $s_i = [y_i, x_i]$, where the y_i obey the recurrence

$$y_i = \begin{cases} a_0 & i = 0 \\ y_{i-1} \odot a_i & 0 < i < n, \end{cases} \tag{8}$$

and the x_i are from (5). Using (5), (6) and (8) we obtain:

$$\begin{aligned} s_0 &= [y_0, x_0] \\ &= [a_0, b_0] \\ &= c_0 \\ s_i &= [y_i, x_i] \qquad\qquad 0 < i < n \\ &= [y_{i-1} \odot a_i,\ (x_{i-1} \otimes a_i) \oplus b_i] \\ &= [y_{i-1} \odot c_{i,a},\ (x_{i-1} \otimes c_{i,a}) \oplus c_{i,b}] \\ &= [y_{i-1}, x_{i-1}] \bullet c_i \\ &= s_{i-1} \bullet c_i. \end{aligned}$$

Since $\bullet$ is associative, we have reduced (5) to (4). The results x_i are just the second values of s_i (the $s_{i,b}$). This allows us to use the scan algorithm of Section 2 with operator $\bullet$ to solve any recurrence of the form (5) on an EREW P-RAM in time:

$$(T_\odot + T_\otimes + T_\oplus) T_S(n,p) = 2(T_\odot + T_\otimes + T_\oplus)(n/p + \lg p) \tag{9}$$

where $T_\odot$, $T_\otimes$ and $T_\oplus$ are the times taken by $\odot$, $\otimes$ and $\oplus$ ($\bullet$ makes one call to each). If all that is needed is the final value x_{n-1}, then we can use a reduce instead of scan with the operator $\bullet$, and the running time is:

$$(T_\odot + T_\otimes + T_\oplus) T_R(n,p) = (T_\odot + T_\otimes + T_\oplus)(n/p + \lg p) \tag{10}$$

which is asymptotically a factor of 2 faster than (9).

Applications of first-order linear recurrences include various time-varying linear systems, the backsubstitution phase of tridiagonal linear-systems solvers, and evaluation of polynomials.

4.2 Higher Order Recurrences

We now consider the more general order m recurrences of the form:

$$x_i = \begin{cases} b_i & 0 \le i < m \\ (x_{i-1} \otimes a_{i,1}) \oplus \cdots \oplus (x_{i-m} \otimes a_{i,m}) \oplus b_i & m \le i < n \end{cases} \tag{11}$$

where $\oplus$ and $\otimes$ are binary operators with the same three restrictions as in (5): $\oplus$ is associative, $\otimes$ is semiassociative, and $\otimes$ distributes over $\oplus$.

To convert this equation into the form (5), we define the following vector of variables:

$$s_i = [\ x_i \ \cdots \ x_{i-m+1}\ ]. \tag{12}$$

a	=	[5	1	3	4	3	9	2	6]
f	=	[1	0	1	0	0	0	1	0]
segmented +-scan	=	[5	6	3	7	10	19	2	8]
segmented max-scan	=	[5	5	3	4	4	9	2	6]

Figure 10: The segmented scan operations restart at the beginning of each segment. The vector f contains flags that mark the beginning of the segments.

Using (11) we can write (12) as:

$$s_i = \begin{bmatrix} x_{i-1} & \cdots & x_{i-m} \end{bmatrix} \otimes_{(v)} \begin{bmatrix} a_{i,1} & 1 & 0 & \cdots & 0 \\ \vdots & 0 & 1 & & \vdots \\ \vdots & \vdots & & \ddots & 0 \\ \vdots & 0 & \cdots & 0 & 1 \\ a_{i,m} & 0 & \cdots & 0 & 0 \end{bmatrix} \oplus_{(v)} \begin{bmatrix} b_i & 0 & \cdots & 0 \end{bmatrix}$$

$$= (s_{i-1} \otimes_{(v)} A_i) \oplus_{(v)} B_i \tag{13}$$

where $\otimes_{(v)}$ is vector-matrix multiply and $\oplus_{(v)}$ is vector addition. If we use matrix-matrix multiply as the companion operator of $\otimes_{(v)}$, then (13) is in the form (5). The time taken for solving equations of the form (11) on an EREW P-RAM is therefore:

$$(T_{m\otimes m}(m) + T_{v\otimes m}(m) + T_{v\oplus v}(m))T_S(n,p) = O((n/p + \lg p)T_{m\otimes m}(m)) \tag{14}$$

where $T_{m\otimes m}(m)$ is the time taken by an $m \otimes m$ matrix multiply. The sequential complexity for solving the equations is $O(nm)$, so the parallel complexity is optimal in n when $n/p \geq \lg p$, but is not optimal in m—the parallel algorithm performs a factor of $O(T_{M\otimes M}(m)/m)$ more work than the sequential algorithm.

Applications of the recurrence (11) include solving recurrences of the form $x_i = a_i + b_i/x_{i-1}$ (see problem 10), and generating the first n Fibonacci numbers $x_0 = x_1 = 1$, $x_i = x_{i-1} + x_{i-2}$ (see problem 11).

5 Segmented Scans

This section shows how the array operated on by a scan can be broken into segments with flags so that the scan starts again at each segment boundary (see Figure 10). Each of these scans takes two arrays of values: a *data* array and a *flag* array. The *segmented scan* operations present a convenient way to execute a scan independently over many sets of values. The next section shows how the segmented scans can be used to execute a parallel quicksort, by keeping each recursive call in a separate segment, and using a segmented +-scan to execute a split within each segment.

The segmented scans satisfy the recurrence:

$$x_i = \begin{cases} a_0 & i = 0 \\ \begin{cases} a_i & f_i = 1 \\ (x_{i-1} \oplus a_i) & f_i = 0 \end{cases} & 0 < i < n \end{cases} \tag{15}$$

where $\oplus$ is the original associative scan operator. If $\oplus$ has an identity $I_\oplus$, then (15) can be written as:

$$x_i = \begin{cases} a_0 & i = 0 \\ (x_{i-1} \times_s f_i) \oplus a_i & 0 < i < n \end{cases} \tag{16}$$

where $\times_s$ is defined as:

$$x \times_s f = \begin{cases} I_\oplus & f = 1 \\ x & f = 0. \end{cases} \tag{17}$$

This is in the form (5) and $\times_s$ is semiassociative with logical **or** as the companion operator (see Problem 9). Since we have reduced (15) to the form (5), we can use the technique described in Section 4.1 to execute the segmented scans in time

$$(T_{\text{or}} + T_{\times_s} + T_\oplus)T_S(n,p)\ . \tag{18}$$

This time complexity is only a small constant factor greater than the unsegmented version since **or** and $\times_s$ are trivial operators.

5.1 Example: Quicksort

To illustrate the use of segmented scans, we consider a parallel version of quicksort. Similar to the standard sequential version, 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 parallel algorithm has an expected time complexity of $O(T_S(n,p)\ \lg n) = O(\frac{n}{p}\lg n + \lg^2 n)$.

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. Figure 11 shows pseudocode for the parallel quicksort and gives an example. The steps of 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 a reduce with logical **and** to check if all the elements are in order.

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

```
procedure quicksort(keys)
  seg-flags[0] ← 1
  while not-sorted(keys)
    pivots    ← seg-copy(keys, seg-flags)
    f         ← pivots <=> keys
    keys      ← seg-split(keys, f, seg-flags)
    seg-flags ← new-seg-flags(keys, pivots, seg-flags)
```

Key	=	[6.4	9.2	3.4	1.6	8.7	4.1	9.2	3.4]
Seg-Flags	=	[1	0	0	0	0	0	0	0]
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]
Seg-Flags	=	[1	0	0	0	1	1	0	0]
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]
Seg-Flags	=	[1	1	0	1	1	1	1	0]

Figure 11: An example of 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. The operation <=> returns one of three values depending on whether the first argument is less than, equal to or greater than the second.

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 scan with the binary operator copy, which returns the first of its two arguments:

   $$a \leftarrow \mathsf{copy}(a, b) .$$

   This has the effect of copying the first element of each segment across the segment. The algorithm could also pick a random element within each segment (see Problem 15).

3. Within each segment, compare each element with 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 3 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 +-scan operation to generate indices relative to the beginning of each segment, and we can use a segmented copy-scan to copy the offset of the beginning of each segment across the segment. We then add the offset to the segment indices to generate the location 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 scans and to the primitives of the P-RAM. If we select pivots randomly within each segment, quicksort is expected to complete in $O(\lg n)$ iterations, and therefore has an expected running time of $O(\lg n \cdot T_S(n, p))$.

The technique of recursively breaking segments into subsegments and operating independently within each segment can be used for many other divide-and-conquer algorithms, such as mergesort.

6 Allocating Processors

Consider the following problem: given a set of processors, each containing an integer, allocate that integer number of new processors to each initial processor. Such allocation is necessary in the parallel line-drawing routine described in Section 6.1. In this line-drawing routine, each processor 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. We can test or search the moves in parallel by placing each possible move in a separate processor. Since the algorithm dynamically decides how many next moves

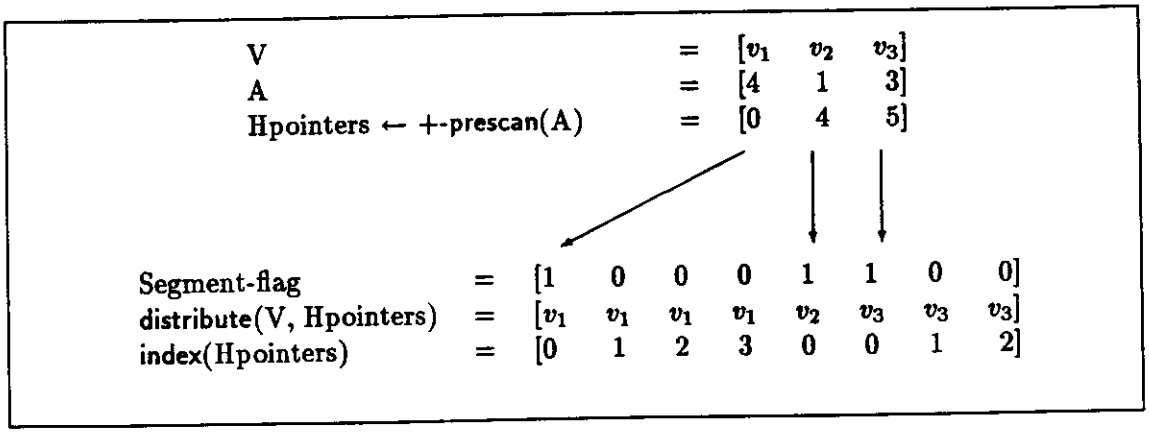


Figure 12: An example of processor allocation. The vector A specifies how many new elements each position needs. We can allocate a segment to each position by applying a +-prescan to A and using the result as pointers to the beginning of each segment. We can then distribute the values of V to the new elements with a permute to the beginning of the segment and a segmented copy-scan across the segment.

to generate (depending on the position), we need to dynamically allocate new processing elements.

More formally, given a length l vector A with integer elements a_i, allocation is the task of creating a new vector B of length

$$L = \sum_{i=0}^{l-1} a_i \tag{19}$$

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 that we can use the same processors for both.

Allocation can be implemented by assigning a contiguous segment of elements to each position i of A. To allocate segments we execute a +-prescan on the vector A that returns a pointer to the start of each segment (see Figure 12). We can then generate the appropriate segment flags by writing a flag to the index specified by the pointer. To distribute values from each position i to its segment, we write the values to the beginning of the segments and use a segmented copy-scan operation to copy the values across the segment. Allocation and distribution each require one call to a scan and therefore have complexity $T_S(l,p)$ and $T_S(L,p)$ respectively.

Once a segment has been allocated for each initial element, it is often necessary to generate indices within each segment. We call this the index operation, and it can be implemented with a segmented `+-prescan`.

6.1 Example: Line Drawing

As an example of how allocation is used, consider line drawing. The line-drawing problem is: given a set of pairs of points $(\langle(x_0, y_0) : (\hat{x}_0, \hat{y}_0)\rangle, \ldots, \langle(x_{n-1}, y_{n-1}) : (\hat{x}_{n-1}, \hat{y}_{n-1})\rangle)$, generate all the locations of pixels that lie between on of the pairs of points. Figure 13 illustrates an example. The routine we discuss returns a vector of (x, y) pairs that specify the position of each pixel along every line. If a pixel appears in more than one line, it will appear more than once in the vector. The routine generates the same set of pixels as generated by the simple digital differential analyzer sequential technique.

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. Figure 13 shows the code. 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 one endpoint along with the per-pixel x and y increments across the segment. 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 endpoint the slope and the position in the line (determined with a `index` operation), each pixel can determine its final (x, y) location in the grid.

This routine has the same complexity as a scan $T_S(m, p)$, where m is the total number of pixels. To actually place the points on a grid, rather than just generating their position, 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.

Exercises

1 Modify the algorithm in Figure 4 to execute a scan instead of a prescan.

2 Use the scan operation to compare two strings of length n in $O(n/p + \lg p)$ time on an EREW P-RAM.

3 Given two vectors of bits that represent nonnegative integers, show how a prescan can be used to add the two numbers (return a vector of bits that represents the sum of the two numbers).

4 Trace the steps of the split-radix sort on the vector

$$[2 \quad 11 \quad 4 \quad 5 \quad 9 \quad 6 \quad 15 \quad 3].$$

5 Show that subtraction is semiassociative and find its companion operator.

6 Write a recurrence equation of the form (5) that evaluates a polynomial

$$y = b_1 x^{n-1} + b_2 x^{n-2} + \cdots + b_{n-1} x + b_n$$

```
procedure line-draw(x, y)
   in parallel for each line i
      % determine the length of the line
      length[i] ← maximum(|p2[i].x − p1[i].x|, |p2[i].y − p1[i].y|)

      % determine the x and y increments
      Δ[i].x ← (p2[i].x − p1[i].x) / length[i]
      Δ[i].y ← (p2[i].y − p1[i].y) / length[i]

   % distribute values and generate index
   p1' ← distribute(p1, lengths)
   Δ' ← distribute(Δ, lengths)
   index ← index(lengths)

   in parallel for each pixel j
      % determine the final position
      result[j].x ← p1'[j].x + round(index[j] × Δ'[j].x)
      result[j].y ← p1'[j].y + round(index[j] × Δ'[j].y)
```

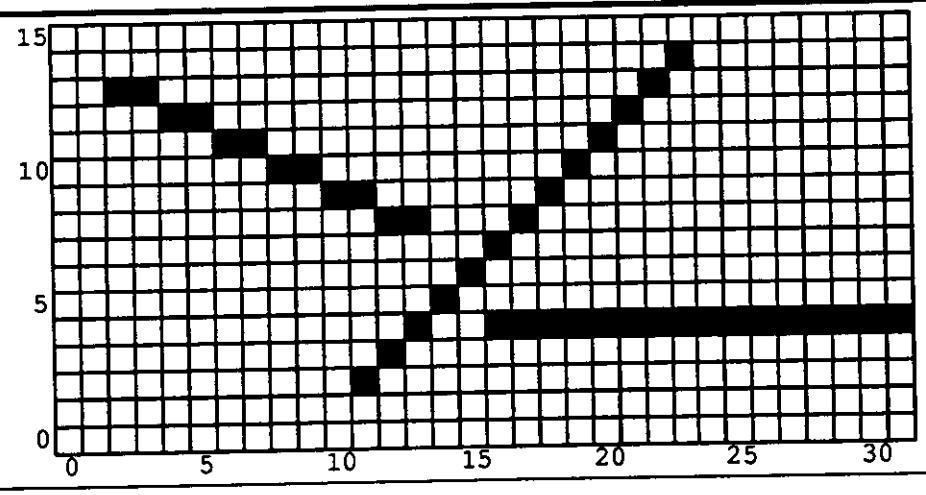


Figure 13: The pixels generated by a line drawing routine. In this example the endpoints are $\langle(11,2):(23,14)\rangle$, $\langle(2,13):(13,8)\rangle$, and $\langle(16,4):(31,4)\rangle$. The algorithm allocates 12, 11 and 16 pixels respectively for the three lines.

for a given x.

7 Show that if $\otimes$ has an inverse, the recurrence of the form (5) can be solved with some local operations (not involving communication among processors) and two scan operations (using $\otimes$ and $\oplus$ as the operators).

8 Prove that vector-matrix multiply is semiassociative.

9 Prove that the operator $\times_s$ defined in (17) is semiassociative.

10 Show how the recurrence $x(i) = a(i) + b(i)/x(i-1)$, where + is numeric addition and / is division, can be converted into the form (11) with two terms (m = 2).

11 Use a scan to generate the first n Fibonacci numbers.

12 Show how to solve a tridiagonal linear-system using the recurrences in Section 4. Is the algorithm asymptotically optimal?

13 In the language Common Lisp, the % character means that what follows the character up to the end of the line is a comment. Use the scan operation to mark all the comment characters (everything between a % and an end-of-line).

14 Trace the steps of the parallel quicksort on the vector

$$[27 \quad 11 \quad 51 \quad 5 \quad 49 \quad 36 \quad 15 \quad 23].$$

15 Describe how quicksort is changed so that it selects a random element within each segment for a pivot.

16 Design an algorithm that given the radius and number of sides on a regular polygon, determines all the pixels that outline the polygon.

Notes

The all-prefix-sums operation has been around for centuries as the recurrence $x_i = a_i + x_{i-1}$. A parallel circuit to execute the scan operation was first suggested by Ofman in 1963 [15] for the addition of binary numbers. A parallel implementation of scans on a perfect shuffle network was later suggested by Stone [17] for polynomial evaluation. The optimal algorithm discussed in Section 2 is a slight variation of algorithms suggested by Kogge and Stone [10] and by Stone [19] in the context of recurrence equations.

Ladner and Fischer first showed an efficient general-purpose circuit for implementing the scan operation [11]. Brent and Kung, in the context of binary addition, first showed an efficient VLSI layout for a scan circuit [4]. More recent work on implementing scan operations in parallel include the work of Fich [7] and of Lakshmivarahan, Yang and Dhall [12], which give improvements over the circuit of Ladner and Fischer, and of Lubachevsky and Greenberg [13], which demonstrates the implementation of the scan operation on asynchronous machines. Blelloch suggested that certain scan operations be included in the P-RAM model as primitives and shows how this affects the complexity of various algorithms [1].

The line-of-sight and radix-sort algorithms are discussed by Blelloch [2, 3]. The parallel solution of recurrence problems was first discussed by Karp, Miller and Winograd [9], and parallel algorithms to solve them are given by Kogge and Stone [10], Stone [18, 19] and Chen and Kuck [5]. Hyafil and Kung [8] show that the complexity (10) is a lower bound.

Schwartz [16] and, independently, Mago [14] first suggested the segmented versions of the scans. Blelloch suggested many uses of these scans including the quicksort algorithm and the line-drawing algorithm presented in Sections 5.1 and 6.1 [2].

Acknowledgments

I would like to thank Siddhartha Chatterjee, Jonathan Hardwick and Jay Sipelstein for reading over drafts of this paper and helping to clean it up.

References

[1] Guy E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[2] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.

[3] 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.

[4] R. P. Brent and H. T. Kung. The Chip Complexity of Binary Arithmetic. In *Proceedings ACM Symposium on Theory of Computing*, pages 190–200, 1980.

[5] Shyh-Ching Chen and David J. Kuck. Time and Parallel Processor Bounds for Linear Recurrence Systems. *IEEE Transactions on Computers*, C-24(7), July 1975.

[6] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, June 1989.

[7] Faith E. Fich. New Bounds for Parallel Prefix Circuits. In *Proceedings ACM Symposium on Theory of Computing*, pages 100–109, April 1983.

[8] L. Hyafil and H. T. Kung. The Complexity of Parallel Evaluation of Linear Recurrences. *Journal of the Association for Computing Machinery*, 24(3):513–521, July 1977.

[9] R. H. Karp, R. E. Miller, and S. Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the Association for Computing Machinery*, 14:563–590, 1967.

[10] 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.

[11] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[12] S. Lakshmivarahan, C. M. Yang, and S. K Dhall. Optimal Parallel Prefix Circuits with (size + depth) = $2n - n$ and $\lceil \log n \rceil \leq depth \leq \lceil 2 \log n \rceil - 3$. In *Proceedings International Conference on Parallel Processing*, pages 58–65, August 1987.

[13] Boris D. Lubachevsky and Albert G. Greenberg. Simple, Efficient Asynchronous Parallel Prefix Algorithms. In *Proceedings International Conference on Parallel Processing*, pages 66–69, August 1987.

[14] G. A. Mago. A network of computers to execute reduction languages. *International Journal of Computer and Information Sciences*, 1979.

[15] Yu. Ofman. On the Algorithmic Complexity of Discrete Functions. *Soviet Physics Doklady*, 7(7):589–591, January 1963.

[16] Jacob T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, October 1980.

[17] Harold S. Stone. Parallel Processsing with the Perfect Shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, 1971.

[18] Harold S. Stone. An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. *Journal of the Association for Computing Machinery*, 20(1):27–38, January 1973.

[19] Harold S. Stone. Parallel Tridiagonal Equation Solvers. *ACM Transactions on Mathematical Software*, 1(4):289–307, December 1975.