LINEAR ALGORITHMS THAT ARE EFFICIENTLY PARALLELIZED TO TIME O(log n)

Ted Herman

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-17 September 1985

Abstract

We consider programming problems characterized by O(n) execution time on sequential machinery and O(log n) execution time on parallel machinery. We demonstrate, for selected problems, efficient parallel algorithms that have O(log n) execution time using O(n/log n) processors. Two specific results include: merging two n-item lists and adding two n-bit numbers.

Key Words. Parallel algorithms, divide-and-conquer, associativity.

This work was supported by Air Force Grant AFOSR 81-0205.

1.0 *Introduction*

Parallel algorithms are interesting not only because they compute results faster, but also because they offer new insights on the nature of the problems they solve. Many problems do not have fast parallel algorithms while others have parallel algorithms requiring an unreasonable number of processors. The present investigation is a catalog of fast parallel algorithms that employ a small number of processors. We have selected programming problems that satisfy two criteria: They have sequential linear-time algorithms and there are published parallel algorithms solve the same problems in time proportional to the logarithm of the input size. Our algorithms also have the logarithmic time, but employ an asymptotically optimum number of processors.

Associativity and the strategem of divide-and-conquer underly the development of the algorithms. Two of our problems have associative functions as input parameters. These problems are paradigms, representing classes of programming problems, and we provide specific examples to demonstrate the forms. We use the divide-and-conquer method repeatedly to achieve O(log n) execution time; the property of associativity enables us to use divide-and-conquer.

The topics of this chapter have the following organization: First, the computing model is defined. A short review of complexity theory for the model follows, which leads to the definition of efficiency. Then, after some notational remarks, the efficient paradigms appear with examples of application. We close with some speculative comments.

1.1 *Computer Model*

Two computer models are used in this paper, a sequential machine model and a parallel machine model. The Random Access Machine model (RAM) is suited to sequential analysis. A RAM has a single processor which utilizes a simple instruction set. Each instruction executes in a constant amount of time and operates on a constant number of memory elements. A chief advantage of the RAM model is the close resemblance to existing computers. RAM algorithms may be easily implemented.

A Parallel Random Access Machine model (PRAM) serves for parallel analysis. A PRAM has many processors, each of which operates on instructions as does a RAM processor. Together, the processors execute in synchrony: If a number of processors execute identical instruction sequences, then they complete the sequences at the same time. Any number of processors may simultaneously examine the same memory element, but no two processors are allowed to simultaneously update a memory element. Algorithms designed for a PRAM guarantee this last condition through timing dependencies, relying on synchrony. There are no "locking" primitives.

The convenience of the PRAM model is its resemblance to traditional sequential models: A PRAM constrained to a single processor is a RAM. Programming for a PRAM is facilitated by the rich inheritance from the RAM framework. The PRAM model, or closely related models, are the most commonly used for parallel complexity results found in the literature. Unfortunately, the PRAM is not as realistic as the RAM model. The existing parallel computers do not closely resemble a PRAM. The PRAM results may be simulated by an appropriate parallel implementation, but the overhead of simulation overhead may raise the time bound significantly.

In both RAM and PRAM models, processors may use address arithmetic in the course of calculations. We assume that simple address computations execute in constant time with respect to the input data size. Permitting address arithmetic raises some sensitive questions in complexity theory [7] that are outside the scope of the present investigation.

The power of address arithmetic underlies many common sorting and searching algorithms. If the input size for a binary search problem is n, an address to an element of the input is represented with (lg n) bits (the function (lg n) is the base two logarithm of n). It is reasonable to claim that adding two (lg n) bit numbers requires time O(lg(lg n)); and in the RAM model, we regard O(lg(lg n)) as a constant with respect to n. Similarly, for a PRAM the O(lg(lg n)) factor is judged constant with respect to O(lg n) time. This last approximation, though asymptotically weaker, extends the usefulness of the paradigms in this paper.

1.2 *Complexity*

Some terminology from complexity theory provides the basis for our definition of efficiency of parallel algorithms. A main issue of complexity theory is the classification of programming problems by their computational resource requirements. The typical measure for classification is a bound on the running time for the fastest known algorithm to solve a problem. The time bound is expressed as a function of the problem's input size. When a faster algorithm is discovered, then the problem is reclassified. Parallel machinery offers the possibility of speeding up running time and consequently has a role in the classification system.

In the analysis of a problem's resource requirements, there are no input and output operations. The input data is given initially in memory, and the result of execution is in memory. Although data structures are permitted in the computer models, the examples in this paper do not resort to any special data structures.

The notation for RAM time complexity is P^i: The fastest sequential algorithms for problems in P^i have time bound $O(n^i)$, where n is the problem's input size. P^1 is the class of linear algorithms.

The notation for PRAM time complexity is NC^k. The fastest PRAM algorithms for problems in NC^k have time bound $O((\lg n)^k)$, where n is the problem's input size. The number of processors, not explicitly specified, is bounded by $O(n^j)$ for some constant j. The classification of programming problems into classes NC^k is an active field of study [1,6]. The relationships between classes of P^i and NC^k are not obvious. There are linear RAM algorithms that do not have NC^k classifications, whereas some problems in P^2 are also NC^1. The NC^k classification of problems is interesting as a new measure of complexity, but it is oblivious to processor requirement (within boundedness).

Problems in NC^k can be further classified by a bound on the number of processors needed to achieve the $O((\lg n)^k)$ time bound. In this paper we define $NC^k(g(n))$ to be the subset of NC^k that requires $O(g(n))$ processors. For example, a PRAM restricted to one processor is a RAM, so $NC^1(1)$ denotes the class of RAM problems that have $O(\lg n)$ time bound; an instance is binary search.

1.3 *Simulation and Efficiency*

PRAM algorithms may be simulated on a RAM machine. Consider B, a problem in $NC^1(g(n))$. The PRAM algorithm to solve B can be simulated on a RAM as follows: For each step of the PRAM algorithm, the RAM machine executes the O(g(n)) instructions corresponding to the PRAM instruction cycle. The single PRAM step takes constant time for O(g(n)) processors; the RAM simulation sequentially executes the corresponding instructions in time O(g(n)). The simulation is accurate because no two instructions in a single PRAM step write the same unit of memory.

The RAM simulation of a problem in $NC^1(g(n))$ takes O(g(n)(lg n)) time. By simulation, it follows that the RAM time complexity P^k for the same problem satisfies

$$O(n^k) \leq O(g(n)(\lg n)). \qquad (1.3.0)$$

When (1.3.0) is an equality then we say $NC^1(g(n))$ *efficiently parallelizes* problem B.

For the remainder of the paper we consider problems known to be in the class $P^1 \cap NC^1$. Each of the problems studied are efficiently parallized, that is, the problems are in $NC^1(n/\lg n)$. It may seem odd to the reader to specify computations that prescribe n/(lg n) processors; however, these are asymptotic results. Therefore it is unimportant that (lg n) or n/(lg n) may not have integral values. In a detailed implementation discussion, floor and ceiling functions would appear with constant factors. These adjustments do not improve the clarity of exposition, and the order of magnitude results are unaffected, so we omit some implementation details for the presentation. Such remarks will also be understood later, when schemes are described that divide the input data into groups of size (lg n) and so forth.

1.4 *Notation*

Our presentation of parallel algorithms uses conventions for defining atoms, sequences and functions. Generally, capital letters are reserved for sequences and lowercase letters for atoms or functions.

Atoms. We regard atoms as indivisible units. Instances of atoms may be single bits in memory, numbers, character strings or matrices, but from the point of view of our notation, the internal structure of an atom is indescribable. In memory, we require that an atom be represented with a constant number of bits.

The symbol for definition is "≈" whereas "=" expresses the comparison of two terms. An atom may be defined directly with an expression, say $x \approx 5+2$, or with a conditional expression with guarded clauses as in [10] . For example,

$$x \approx \textbf{if } y<0 \rightarrow 0 \;\Box\; y=0 \rightarrow 1 \;\Box\; y>0 \rightarrow 2 \textbf{ fi}$$

defines x to have the value 0, 1, or 2 if y is negative, zero or positive (respectively).

Sequences. The notation for sequence construction uses square brackets. If x is an atom, then [x] is a sequence with one element, x. Let x and y be atoms, and define C to the sequence [x y y x] , a sequence of four elements. C.*i* refers to the *i*-th element of C (we start the indexing with zero). Thus C.0 and C.3 have the value x.

Let D be a sequence of n atoms. A notation for sequence D is " [range: prototype expression] ," exemplified as follows:

$$D \approx [\ 0 \leq i < n\text{: } D.i\ ] .$$

The prototype expression includes a dummy variable defined in the range. A sequence E consisting of D's elements of even index is:

$$E \approx [\ 0 \le j < (n/2):\ D.(2j)\ ]\ .$$

The following formula describes a sequence F that maps D's negative elements to the value zero.

$$F \approx [\ 0 \le i < n:\ \mathbf{if}\ D.i > 0 \rightarrow D.i\ \square\ D.i \le 0 \rightarrow 0\ \mathbf{fi}\ ]\ .$$

A sequence is *empty* if it has no elements. Two examples of empty sequences are: [ ] , and

$$[\ n \le i < n:\ G.i\ ]\ .$$

Let $|H|$ denote the number of elements in a sequence H; $|H| = 0$ for the empty sequence.

We shall not consider sequences whose elements are, in turn, sequences. The operation of sequence construction is therefore idempotent: $[\ [\ x\ ]\ ] = [\ x\ ]$. Consider the two sequences A and B:

$$A \approx [\ 0 \le i < n:\ A.i\ ]\text{, and } B \approx [\ 0 \le i < m:\ B.i\ ]\ .$$

Then let $C \approx [\ A\ B\ ]$. An equivalent definition of C is:

$$C \approx [\ 0 \le i < (n+m):\ \mathbf{if}\ i < n \rightarrow A.i\ \square\ i \ge n \rightarrow B.(i\text{-}n)\ \mathbf{fi}\ ]\ .$$

Functions. A function is a value or a sequence defined by a formula that equates the name of the function and bound variables with a constructor. Examples: **bump** is a function to

add one to its argument: bump.x ≈ x + 1. Function **vectorbump** adds one to each item in a sequence:

vectorbump.X ≈ [$0 \le i < |X|$: bump.(X.i)] .

The greatest common divisor function is defined recursively:

gcd.a.b ≈ **if** a=b→ b ▯ a<b→ gcd.a.(b-a) ▯ a>b→ gcd.(a-b).b **fi**.

In this example, the name **gcd** appears in two senses; the prototype is just left of the "≈" and the recursive references to **gcd** are in the expression to the right of "≈."

Some especially useful functions for dealing with sequences are **head**, **tail**, and **last**.

head.D ≈ **if** |D|=0→ [ ] ▯ |D|>0→ D.0 **fi**.

tail.D ≈ [$1 \le i < |D|$: D.i] .

last.D ≈ **if** |D|=0→ [ ] ▯ |D|>0→ D.(|D|-1) **fi**.

Inspection of the definitions shows that D = [head.D tail.D] .

In this paper the computation of a function has no side-effects. The result of function evaluation is an atom or a sequence that, for our purposes, is stored at some virgin location in memory -- memory never used before the function evaluation. In contrast, a side-effect computation (assignment statement) stores an atom in memory at an old location, one that has previously been defined by a function or contains some initial input. It is more difficult to show correctness of parallel computations with side-effects. With assignment statements it is possible to design a program specifying simultaneous writes to one memory location;

it is vital that specifications of parallel assignments be pre-conditioned to avoid such violation of PRAM restrictions. In fact, we avoid the use of side-effects whenever possible. The justification for side-effects is efficiency of implementation.

After circumspect isolation and demonstration of required side-effect computations, we shall use an equivalent functional definition. Furthermore, we label the side-effect to attract the reader's attention:

Side-Effect(C.i := x)

denotes the computation that dynamically changes the definition of element C.i in the sequence C. Suppose P is a permutation of the indices in the range $0 \leq i < n$. A "function" to invert P is given by

Invert.P ≈ C,

Side-Effect([$0 \leq i < n$: C.(P.i) := i]).

The assignments may be parallel because all atoms in P are distinct and within the appropriate range.

Functionals. A function may have function names as arguments. Given functions f and g, the expression f.g.x is to be interpreted as f with the arguments g and x. The expression f.(g.x) refers to f with argument g.x. And (f.g).x is the *function* f.g with argument x.

Associativity. The symbol ★ is a binary function with domain and range in some set A. The system (A,★) is a monoid [9], that is:

(i) For all a,b∈A, ★.a.b∈A.

(ii) For all a,b,c∈A, ★.(★.a.b).c = ★.a.(★.b.c).

(iii) There is an identity element $\mathcal{I}$∈A satisfying: For all a∈A, {★.a.$\mathcal{I}$ = ★.$\mathcal{I}$.a *and* ★.a.$\mathcal{I}$ = a}.

1.5 *Parallel Paradigms*

Each of the paradigms introduced in this section is associated with problem that is efficiently parallelized. For each problem, there is a description of the problem's classification in P^1 and NC^1. Then we show the problem to be in the class NC^1(n/lg n). Examples that fit the paradigm are cited to show the generality of the paradigm.

For the paradigms that follow, let **IN** be the sequence of n>0 atoms that hold the input for a programming problem: IN ≈ [0≤*i*<n: IN.*i*] .

1.5.0 *Function Distribution*

We begin with a simple paradigm called function distribution, or **apply**. With this problem, definitions and methods appear that will be useful in exposing the paradigms in later sections. Let *f* be a function satisfying

(i) *f* maps atoms to atoms, and

(i) For each *i*, 0≤*i*<n, *f*.(IN.*i*) is computable in constant RAM time.

The problem is to compute, from the input sequence and function *f*, the output sequence given by the function **apply**:

$$\text{apply}.f.\text{IN} \approx [\ 0 \le i < n: f.(\text{IN}.i)\ ]\ . \qquad (1.5.0)$$

1.5.0.0 Problem (1.5.0) is in P^1.

Demonstration. An alternative definition of **apply** has recursive form:

$$\text{apply}.f.\text{A} \approx \textbf{if}\ |\text{A}|<1 \rightarrow [\ ]\ \square\ |\text{A}|>0 \rightarrow [\ f.(\text{A}.0)\quad \text{apply}.f.(\text{tail}.\text{A})\ ]\ \textbf{fi}.$$

The equivalence of the recursive definition and (1.5.0) is easily shown by induction. It also follows by induction, from this recursive definition, that a simple RAM iteration solves **apply** in time O(n). ■

1.5.0.1 Instead of showing (1.5.0) is in NC^1, we have the result: Problem (1.5.0) is in $NC^0(n)$.

Demonstration. The definition (1.5.0) may be equivalently written

$$\text{apply}.f.\text{IN} \approx \text{C},$$

$$\textit{Side-Effect}(\ [\ 0 \le i < n: \text{C}.i := f(\text{IN}.i)\ ]\ ).$$

By assigning n processors, on processor to each atom of IN, all computations finish in constant time. The processors store results in parallel with no memory write conflicts, so the time complexity is constant with O(n) processors. ■

1.5.0.2 Problem (1.5.0) is in $NC^1(n/\lg n)$.

Demonstration. Let **group** be a function of sequence D and integers p and k, such that $0<p<|D|$ and $0 \le k < |D|/p$,

$$\text{group}.\text{D}.\text{p}.\text{k} \approx [\ 0 \le i < p: \text{D}.(k*p+i)\ ]$$

Informally, **group** divides D into (|D|/p) groups of equal size: "group.D.p.k" refers to the k-th group, a subsequence of p elements. If (|D| mod p)=0, then

$$D = [\ 0 \le i < |D|/p: \text{group.D.p.}i\ ] .$$

Similarly, apply.f.D may be written in the recursive form

$$\text{apply.}f\text{.D} = [\ 0 \le i < |D|/p: \text{apply.}f\text{.(group.D.p.}i)\ ] \qquad (1.5.0.2)$$

The formulation (1.5.0.2) expresses the divide-and-conquer strategem. Just as reworking the definition of **apply** to the recursive form in 3.5.0.0 suggested the RAM program, this new formulation will suggest a PRAM program. The computation of **apply** reduces to subproblems of **apply**, each of p elements. An instance of a subproblem "apply.f.(group.D.p.i)" can be computed sequentially by a single processor, that is, in time O(p) according to 1.5.0.0. With (|D|/p) processors, one processor dedicated to each subsequence, all subproblems are computed in time O(p). The choice p=(lg n) establishes the efficient classification NC^1(n/lg n). ∎

1.5.1 *Applications of Function Distribution*

The **vectorbump** function defined in 1.4.1 can be written in terms of **apply**: vectorbump.X ≈ apply.bump.X. Therefore adding a constant to a vector (sequence of numbers) is an NC^1(n/lg n) operation. ∎

Variations of **apply** fit other applications. For example, let **apply2** be defined for two sequences A and B, |A|=|B| and |A|=n, and a binary function f,

$$\text{apply2.}f\text{.A.B} \approx [\ 0 \le i < n: f\text{.(A.}i\text{).(B.}i)\ ] .$$

The same arguments that classified **apply** also place **apply2** in P^1, $NC^0(n)$, and $NC^1(n/\lg n)$.

Let **plus** be defined for two atoms: plus.x.y ≈ x+y. The addition of two vectors A and B is apply2.plus.A.B. Similarly **apply2** can be used to compute the **or**, **and**, or **xor** (the exclusive-or) of two bit strings. Some problems of data arrangement and selection fall in the **apply2** paradigm. ∎

The variation **apply3** also uses *f* as a function of two arguments.

$$\text{apply3}.f.A \approx [\ 0 \leq i < n:\ \textbf{if}\ i=0 \rightarrow f.\delta.(A.i)\ [\!]\ i>0 \rightarrow f.(A.(i\text{-}1)).(A.i)\ \textbf{fi}\ ],$$

where δ is some special atom. Variants of the **apply3** paradigm can be used to calculate splines and other sequences that result from functions of adjacent atoms in A. ∎

1.5.2 *Associative Reduction*

Some of the earliest known possibilities for speedup by parallel processing were derived from syntactic analysis of FORTRAN statements. A statement such as

```
PROD = BIK*DEG*B2*C
```

could be executed in parallel: The terms (BIK*DEG) and (B2*C) are computed simultaneously, and a final multiplication yields the result. The stategy behind the parallel evaluation is essentially divide-and-conquer applied to the expression. However, the use of this technique is not applicable to

$DIV = BIK/DEG/B2/C$

because division is not associative. The following paradigm generalizes the observation about parallel evaluation of associative expressions.

Let (A,★) be a monoid as defined in section 1.4, for some set of atoms A, satisfying: For all a,b∈A, ★.a.b is computable in constant RAM time.

Associative reduction, which we abbreviate to **reduce**, is the problem of computing

reduce.★.IN, where **reduce** has the definition

reduce.★.D ≈ **if** |D|=0→ ϑ ▯

|D|>0→ ★.(D.0).(reduce.★.(tail.D)) **fi**. (1.5.2)

An example of **reduce** is "reduce.plus.A" to compute the sum of the atoms in A.

1.5.2.0 Problem (1.5.2) is in P^1.

Demonstration. A simple RAM iteration with an accumulator solves (1.5.2). The time bound O(n) is easily shown by induction on |D|. ■

1.5.2.1 Problem (1.5.2) is in $NC^1(n)$.

Demonstration. The following decomposition of **reduce** is possible for |D|>0 and (|D| mod 2)=0:

reduce.★.D = reduce.★. [$0 \le i < |D|/2$: ★.(D.(2i)).(D.(2i+1))] (1.5.2.1)

In (1.5.2.1) the computation ★ for adjacent atoms in D precedes the application of **reduce**. The sequence of ★ computations results in a sequence of |D|/2 atoms, and may be computed

in $NC^0(n)$ according to 1.5.0.1 with an appropriate variant of **apply**. The validity of (1.5.2.1) may be proved using the associativity of ★ and induction. We extend equation (1.5.2.1) to the recursive definition

$$\text{reduce.}\star.D \approx \textbf{if } |D|=2 \rightarrow \star(D.0).(D.1) \ \square$$
$$|D|>2 \rightarrow \text{reduce.}\star.\ [\ 0 \leq i < |D|/2:\ \star.(D.(2i)).(D.(2i+1))\ ]\ \textbf{fi}.$$

The recursive definition is equivalent to (1.5.2) provided that $|D|=2^k$ for some integer $k>0$. Cases where n is not a power of 2 may be treated, for example, by padding the input to the nearest power of two with ★-identity elements, $\mathcal{I}$. By induction on k it follows that **reduce** is computed in time O(k), since each recursive step is an $NC^0(n)$ computation. The overall bound is $NC^1(n)$. ∎

1.5.2.2 Problem (1.5.2) is in $NC^1(n/\lg n)$.

Demonstration. An early application of this result appears in [3] and the associative reduction paradigm was shown to be $NC^1(n/\lg n)$ in [4] . We assume n is a power of two such that n is divisible by lg n. Then problem reduce.★.IN may be decomposed as follows:

$$\text{reduce.}\star.\text{IN} = \text{reduce.}\star.\ [\ 0 \leq i < (n/\lg n):\ \text{reduce.}\star.(\text{group.IN.}(\lg n).i)\ ]\ .$$

The correctness of the decomposition follows from associativity of ★. The outermost reduction has, for its argument, a sequence of (n/lg n) atoms. By 1.5.2.1 this outermost reduction can be computed in time O(lg(n/lg n)) using O(n/lg n) processors. The innermost reduction is over a sequence of (lg n) atoms; there are (n/lg n) cases of the inner reduction. By 1.5.2.0, each of the inner reductions may be computed in time O(lg n) using one processor. In parallel, all of the inner reductions may be computed in time O(lg n) using O(n/lg n) processors. The overall classification is O(lg n) time using O(n/lg n) processors. ∎

1.5.2.3 A pipeline construction in NC^1(n/lg n) solves problem (1.5.2).

Demonstration. We propose to arrange O(n/lg n) processors in a tree. The tree is a balanced, regular binary tree with a processor at each vertex. The path length from root to leaf is the same for any leaf of this tree. There are n/(2*lg n) leaf processors, so the total number processors in the tree is (n/lg n)-1. We add an additional processor called the *accumulator*, which brings the total number processors to (n/lg n).

The operation of this network is most easily described in terms of synchronous time units, called *beats*. Each beat is a constant amount of time in which a processor completes one computation. During a beat, the function of a non-leaf processor is to compute ★.a.b, where a and b are values calculated during the previous beat by the processor's children in the tree; for the initial beat (for which there is no previous beat), we assume the values of the children, a and b, are $\mathcal{I}$, the identity element for ★.

Also during a beat, leaf processors calculate ★ for two elements of IN. During beat *i*, $0 \le i < \lg n$, the leaf processors will use "group.IN.(n/(lg n)).*i*" as input. At beat (lg n), and after, all leaf processors will calculate ★.$\mathcal{I}$.$\mathcal{I}$. The function of the accumulator at each beat is to compute ★.a.b, where a is the value computed by the accumulator during the previous beat (a=$\mathcal{I}$ at beat zero), and b is the value computed at the tree's root during the previous beat.

At beat (lg n), the sequence IN has been "fed into" the tree. After an additional lg(n/(lg n)) beats, the result "reduce.★.IN" appears at the accumulator.

This scheme has a practical advantage: It is suited to a message based synchronous model appropriate for VLSI design. The tree configuration can be considered as a systolic pipeline, which is an architecture more easily implemented in hardware than a PRAM. ■

1.5.2 *Applications of Associative Reduction*

The sum of a sequence may be computed as "reduce.plus.IN." Other obvious choices for $\star$ are minimum, maximum, multiply, and logical operations. Existential and universal quantifiers are expressed with **reduce**. For example, the system (or,{true,false}) is a monoid with identity element "false." The predicate

$$\exists i,\ 0 \le i < |D|,\ D.i = 0$$

is equivalently stated in our notation as

$$\text{reduce.or.}\ [\ 0 \le i < |D|:\ D.i = 0\ ]\ .$$

Such an application of **reduce** is useful for solving a problem that has existential quantifiers in its specification.

Below, we show how the **reduce** paradigm also applies to some search problems and to computing the solution to a linear recurrence.

Linear Recurrence

Consider a set of linear recurrences for i, $2 \le i \le n$, defining x_i, given the values a_i and b_i:

$$x_i = a_i * x_{i-1} + b_i * x_{i-2},$$

and x_0 and x_1 are inputs. The problem is to compute the value of x_n. The same recurrence relation in matrix form, is

$$\begin{vmatrix} x_i \\ x_{i-1} \end{vmatrix} = \begin{vmatrix} a_i & b_i \\ 1 & 0 \end{vmatrix} \begin{vmatrix} x_{i-1} \\ x_{i-2} \end{vmatrix}$$

which we capture as a vector recursion with the formula

$$\mathbf{y}_i = \mathbf{C}_i \mathbf{y}_{i-1},$$

where $\mathbf{C}_i$ is a matrix containing a_i and b_i as above, and $\mathbf{y}_i$ is the vector of x_i and x_{i-1}. In this context we regard the vectors and matrices as atoms to be dealt with by the parallel reduction paradigm. Let "**matmul**" be a function multiplying two matrices (or a matrix and a vector). Matrix multiplication associates, so associative reduction is possible. Solving for the vector $\mathbf{y}_n$, that is, repeated substitution of the above formula, leads to the result

$$\mathbf{y}_n = \text{matmul.D.}\mathbf{y}_1, \ \text{D} \approx \text{reduce.matmul.} [\ n \geq i \geq 2 \text{: } \mathbf{C}_i\ ] .$$

The computation of y_n (and therefore x_n) is in $NC^1(n/\lg n)$. ∎

Search

The following application illustrates associative reduction for a search problem. After defining the problem, we use classical divide-and-conquer techniques to derive an associative function, which establishes efficient parallelization.

IN for this problem is a sequence of bits. The problem is to determine the length of the longest sequence of zero bits contained in IN. In general, important features of the divide-and-conquer methodology are:

(i) A non-trivial problem is divided into smaller subproblems.

(ii) The solution to a trivial subproblem may be computed in constant RAM time.

(iii) Merging two subproblem results takes constant RAM time.

Also typical of the methodology is some generalization of the original problem statement to facilitate subdivision.

We reason as follows: Divide IN into halves L and R. It is the case that either the longest zero sequence lies totally in L, totally in R, or spans L and R. The former cases submit to recursion, while the latter means that the L partition terminates with a sequence of zeros and the R partition begins with a sequence of zeros. To be able to merge the results of subproblems L and R, we propose the following generalization of the longest zero sequence problem: Determine for a given sequence D of bits, the following three lengths: (1) The longest zero sequence in D; (2) The longest zero sequence that is left-adjacent, that is, it begins at the first bit in D; (3) The longest right adjacent zero sequence in D, that is, it terminates at the last bit in D.

We call the new problem statement **bigzero**. The role of ★ for this application is to merge solutions to subproblems. Thus the solution for sequence IN, having been divided into L and R, is

bigzero.IN = ★.(bigzero.L).(bigzero.R).

Below, we propose an implementation of **bigzero** and ★ that satisfy constant RAM time and associativity constraints. Also, "bigzero.D" is trivial when $|D| \leq 1$. To acheive the $NC^1(n/\lg n)$ classification, we first compute the sequence "apply.bigzero.IN" which is an $NC^1(n/\lg n)$ problem of computing the trivial **bigzero** values for all bits of IN. Then

bigzero.IN = reduce.★.(apply.bigzero.IN)

finishes efficient parallel paradigm.

Implementing Bigzero

For the paradigm, the ★ function maps pairs of atoms to atoms. To implement ★ and **bigzero**, a design of the internal structure of an atom is appropriate.

For D, a sequence of bits, the function "bigzero.D" may be represented as the 4-tuple **(length,left,right,longest)**, where "length.D" is |D|, "left.D" is the size of the longest left-adjacent zero sequence in D; examples are: left. [00110001] =2; left. [10110001] =0. "right.D" is the size of the longest right-adjacent zero sequence in D. "longest.D" is the size of the longest sequence of zeros in D.

The trivial cases of **(length,left,right,longest)** may be computed in constant RAM time. The values are all zero for an empty sequence, and for example, if x is a single bit, then the trivial definition of **left** is ¬x, that is,

left.x ≈ **if** x=0→ 1 ▯ x=1→ 0 **fi**.

A remaining detail is the computation of ★ in constant RAM time. Let A and B be two sequences of bits and let C = [A B] . Then

bigzero.C = ★.(bigzero.A).(bigzero.B)

and ★ is composed of the following definitions:

length.C ≈ length.A + length.B

left.C ≈ **if** (left.A)=(length.A)→ left.A+left.B ▯
(left.A)≠(length.A)→ left.A **fi**.

right.C ≈ **if** (right.B)=(length.B)→ right.A+right.B ▯
(right.B)≠(length.B)→ right.B **fi**.

longest.C ≈ max.(longest.A).(longest.B).(right.A+left.B).

Each of the components of ★ is a constant RAM time calculation, so ★ inherits the constant RAM time bound. The definitions also satisfy the associativity of ★ and have empty sequence as identity element. ∎

1.5.3 *Associative Scan*

The **scan** problem consists of computing a sequence of reductions. For example, "reduce.plus. [0 1 2 3 4 5] " is an atom with value 15; "scan.plus. [0 1 2 3 4 5] " is a the sequence " [0 1 3 6 10 15] ." Each atom in the result of scan is a reduction of a subsequence of the input. The **reduce** problem appears in the definition of the **scan**:

$$\text{scan.}\bigstar\text{.IN} \approx [\ 0 \leq i < \text{n: reduce.}\bigstar\text{. } [\ 0 \leq j < i\text{: IN.}j\ ]\ ] \qquad (1.5.3).$$

The last atom of the result of **scan** is a reduction over the input, that is,

$$\text{last.(scan.}\bigstar\text{.IN)} = \text{reduce.}\bigstar\text{.IN.}$$

1.5.3.0 Problem (1.5.3) is in P^1.

Demonstration. An alternative definition of **scan** has recursive form:

$$\text{scan.}\bigstar\text{.D} \approx \textbf{if } |D| \leq 1 \rightarrow D\ \square\ |D| > 1 \rightarrow [\ E \quad \bigstar\text{.(last.E).(last.D)}\ ]\ \textbf{fi},$$

$$E \approx \text{scan.}\bigstar\text{. } [\ 0 \leq i < |D|\text{-1: D.}i\ ]\ .$$

By induction on n, the equivalence of this definition and (1.5.3) may be shown. The recursive definition expresses the relationship between consecutive elements of the result. Informally, an atom of the result sequence is the ★ calculated for the corresponding input atom and the

reduction of input atoms to its left. Induction on $|D|$ proves the O(n) time bound for computing **scan**. ∎

1.5.3.1 Problem (1.5.3) is in NC^1.

Demonstration. The definition (1.5.3) specifies n reductions, the largest of which is a sequence of n atoms. If all reductions are computed in parallel, then by 1.5.2.2 they can be computed in O(lg n) time. The total processor requirement is O(n*(n/lg n)). ∎

1.5.3.2 Problem (1.5.3) is in $NC^1(n)$.

Demonstration. This was previously shown in [11]. Consider the recursive definition of **scan** given by

$$
\begin{aligned}
\text{scan}.\star.D \approx \textbf{if } & |D|\leq 1 \rightarrow D \ \square \\
& |D|>1 \rightarrow [\,0\leq i<|D|:\ \textbf{if } i=0 \rightarrow D.i \ \square \\
& \qquad (i>0)\wedge((i \bmod 2)=0) \rightarrow \star.(F.((i/2)-1)).(D.i) \ \square \\
& \qquad (i>0)\wedge((i \bmod 2)=1) \rightarrow F.((i-1)/2) \ \textbf{fi}\ ] \\
\textbf{fi},
\end{aligned}
$$

$$F \approx \text{scan}.\star.[\,0\leq j<|D|/2:\ \star.(D.(2j)).(D.(2j+1))\ ].$$

As in the demonstration 1.5.2.1, which established $NC^1(n)$ for reduction, the recursive definition above is equivalent to (1.5.3) provided that $|D|=2^k$ for some integer k>0; when $|D|$ is not a power of two, a padding step can precondition the input. The argument for correctness is inductive: If the computation of F contains results for all atoms of odd index, then **scan** is correctly computed. To compute F, a sequence is generated by **apply** in $NC^0(n)$; there remains a **scan** problem of half the input size, so the O(lg n) time bound follows inductively.

After computing F, the other calculations submit to **apply** and the $NC^0(n)$ classification. The overall time bound is O(lg n) using O(n) processors. ∎

1.5.3.3 Problem (1.5.3) is in $NC^1(n/\lg n)$

Demonstration. We assume n is divisible by (lg n). The decomposition of **scan** is

scan.★.IN = [$0 \le i < n/\lg n$: scan.★.(group.P.(lg n).*i*)] .

We intend the inner reference to **scan**, which has input size (lg n), will be computed by 1.5.3.0 in time O(lg n) by a single processor. Together, all n/(lg n) cases of the inner **scans** may be computed in O(lg n) time using O(n/lg n) processors. The sequence P has n atoms derived from IN: P duplicates IN except that "P.(m.*(lg n))" is a reduction over the sequence " [$0 \le i \le (m*(\lg n))$: IN.*i*] ." This property of P justifies the decomposition above. We now show how P is computed in $NC^1(n/\lg n)$, in two steps. The first step initializes P to IN, which is an $NC^1(n/\lg n)$ computation by **apply**:

Side-Effect([$0 \le i < n$: P.*i* := IN.*i*]).

The second step is

Side-Effect([$0 < i < n/(\lg n)$: P.(*i**(lg n)) := ★.(P.(*i**(lg n))).(F.(*i*-1))]),

which is in $NC^1(n/\lg n)$ if the computation of F is also $NC^1(n/\lg n)$. An element of F is a reduction of a subsequence of IN:

F.m = reduce.★. [$0 \le i < (m+1)*(\lg n)$: IN.*i*] .

Our efficient definition of F is

F ≈ scan.★. [$0 \le i < n/\lg n$: reduce.★.(group.IN.(lg n).*i*)] .

The correctness of F may be established by induction. Each **reduce** computation can be computed in O(lg n) time by one processor. All reductions can be computed in NC^1(n/lg n). The **scan** input for F has n/(lg n) atoms, and by 1.5.3.2 may be computed in O(lg(n/lg n)) time using O(n/lg n) processors. The overall bound for computing F is O(lg n) time with O(n/lg n) processors. ∎

1.5.2 *Applications of Associative Scan*

Many linear search and selection problems can be solved with the **scan** paradigm. We consider two **scan** examples in this section, polynomial evaluation and binary addition.

Polynomial Evaluation

Let P.A.x be a polynomial of degree (n-1). Given a vector of coefficients A, and some value x, the problem is to compute

$$P.A.x \approx \text{reduce.plus.} [\ 0 \leq i < n:\ \text{times.}(A.i).(\text{reduce.times.} [\ 0 \leq j < i:\ x\ ]\ )\ ]\ .$$

Using the **scan** paradigm, the formulation is

$$P.A.x \approx \text{reduce.plus.(apply2.times.A.manyx)},$$

$$\text{manyx} \approx \text{scan.times.} [\ 1\ \ [\ 0 \leq j < (n\text{-}1):\ x\ ]\ \ ]\ .$$

The argument of **scan**, which is the value "1" followed by (n-1) "x" values, can be created by **apply**. Computing "manyx" is then a problem in NC^1(n/lg n). The **apply2** and **reduce** are also of this classification, so we have demonstrated efficient parallelization for polynomial evaluation. ∎

Binary Addition

The inputs for this problem are A and B, both n-bit numbers. The problem is to compute C = A + B. The representation we choose places the least significant bit first: C.0 is the least significant bit and C.(n-1) is the most significant bit. The bit C.0 is easy to calculate:

$$C.0 \approx \text{xor}.(A.0).(B.0),$$

xor being the exclusive-or function. Calculation of other bits of the result requires a carry-in value:

$$C \approx [\,0 \le i < n: \text{xor}.(\text{xor}.(A.i).(B.i)).(\text{carry}.i)\,],$$

where **carry** is recursively defined, for the range $[\,0 \le j < n\,]$:

$$\text{carry}.A.B.j \approx \textbf{if } j=0 \rightarrow 0 \;\square$$

$$j \ne 0 \rightarrow (A.(j\text{-}1) \wedge B.(j\text{-}1)) \vee (A.(j\text{-}1) \wedge (\text{carry}.A.B.(j\text{-}1)) \vee (B.(j\text{-}1) \wedge (\text{carry}.A.B.(j\text{-}1)) \textbf{ fi}.$$

The **carry** function may be viewed as an n-bit vector. If the **carry** vector were calculated in $NC^1(n/\lg n)$ then result C will be computed in $NC^1(n/\lg n)$ by the **apply** paradigm. Therefore we attend to the computation of **carry**.

From its definition, observe that for some cases, "carry.A.B.j" can be calculated regardless of the value of "carry.A.B.(j-1)," that is,

$$A.(j\text{-}1) \wedge B.(j\text{-}1) \Rightarrow \text{carry}.A.B.j, \textit{ and}$$

$$\neg A.(j\text{-}1) \wedge \neg B.(j\text{-}1) \Rightarrow \neg \text{carry}.A.B.j$$

hold for j in the range $[\,1 \le j < n\,]$.

Let **precarry** be the n-bit vector

precarry.A.B ≈ [$0 \leq i < n$: **if**

$i=0 \rightarrow 0$ ▯

$i \neq 0 \rightarrow$ **if**

$\neg A.(i\text{-}1) \wedge \neg B.(i\text{-}1) \rightarrow 0$ ▯

$A.(i\text{-}1) \wedge B.(i\text{-}1) \rightarrow 1$ ▯

$A.(i\text{-}1) \wedge \neg B.(i\text{-}1) \rightarrow 2$ ▯

$\neg A.(i\text{-}1) \wedge B.(i\text{-}1) \rightarrow 2$ **fi**

fi] .

The value "2" for an element of **precarry** represents an unresolved carry bit.

Let ★ be a function over the domain {0,1,2,𝒮}. The definition of ★ is given by:

★.a.b ≈ **if** $b=0 \rightarrow 0$ ▯ $b=1 \rightarrow 1$ ▯ $b=2 \rightarrow a$ **fi.**

The identity element, 𝒮, plays no role in this algorithm; the value 2 is a right identity element of ★.

That ★ is associative is left to the reader. We also omit the proof of the fact that for [$0 \leq j < n$] :

carry.A.B.j = (scan.★.(precarry.A.B)).j

Computing "precarry.A.B" is in NC^1(n/lg n) by **apply**. **scan** is in NC^1(n/lg n), which computes **carry** efficiently in O(lg n) time. ■

1.5.6 *Merging*

In this section we consider the problem of merging sequences. The efficient parallel algorithm is sophisticated and so it is exposed in refinements. Some simplifying assumptions also improve the discussion: We are given two sequences A and B to be merged. A and B are a strictly ascending sequences of n atoms. Also, all atoms in the sequence [A B] have distinct values. For the exposition of **merge**, we use *side-effects*; the **merge** of A and B into C starting at k is

Side-Effect(merge.k.C.A.B),

by which we intend that the subsequence

[$k \leq i < k+2n$: C.i]

will contain the **merge** of A and B.

It is well known that **merge** is in P^1. Batcher's merge [12] shows the problem to be NC^1. A simple demonstration follows.

1.5.6.0 The merge problem is in NC^1.

Demonstration. Let **place** be a function to determine where a specific atom would appear if merged with a given ascending sequence:

place.k.A.B ≈ reduce.plus. [$0 \leq i < |B|$: **if** A.k≥B.i→ 1 ▯ A.k<B.i→ 0 **fi**] .

Computing **place** is, in general, a problem in NC^1(n/lg n). However, B is ascending, so **place** may be computed by a binary search, which is NC^1(1). "B.(place.k.A.B)" is, in general, the smallest value in B larger than "A.k" (the only exception is the case of "A.k" being larger than all atoms in B).

$$\text{merge.k.C.A.B} \approx \textit{Side-Effect}(\; [\, 0 \le i < n:\ C.(k+i+\text{place}.i.A.B) := A.i,\ C.(k+i+\text{place}.i.B.A) := B.i \,]\;).$$

The preconditions on A and B insure that no write conflict occurs if all assignments execute in parallel. The computation of **place**, in parallel, for all 2n cases is a problem in $NC^1(n)$. The assigning is in $NC^1(n/\lg n)$, so the overall classification is $NC^1(n)$. ∎

1.5.6.1 Divide-and-Conquer

Merging is an associative operation, which has the null sequence as identity element. However, the **merge** function does not have a constant RAM time bound, so it differs from the ★ functions considered earlier. By careful division into subproblems, **merge** subproblems can be computed independently with no need to combine results. We illustrate with a simple division.

Consider the four sequences LA, LB, RA and RB,

$$LA \approx [\, 0 \le i < n/2:\ A.i \,],$$

$$LB \approx [\, 0 \le i < (\text{place}.(n/2).A.B):\ B.i \,].$$

$$RA \approx [\, (n/2) \le i < n:\ A.i \,],$$

$$RB \approx [\, (\text{place}.(n/2).A.B) \le i < n:\ B.i \,].$$

With this division, the merge of A and B, may be written

$$\text{merge.k.C.A.B} = [\ \text{merge.k.C.LA.LB} \quad \text{merge.}(k+|LA|+|LB|).\text{C.RA.RB}\ ].$$

To show the decomposition of **merge** is correct, there are three obligations:

(i) Each atom of the input is contained in the input to exactly one subproblem.

(ii) Any atom not contained in a subproblem is either smaller than or greater than every atom in that subproblem.

(iii) The output of the **merge** subproblem will be stored at the proper location in C.

Condition (i) is trivially met by the definitions of LA, LB, RA and RB. Since all atoms in [RA RB] are greater than atoms in [LA LB], condition (ii) is satisfied. The third obligation is upheld because each subproblem stores its result in C starting at (k+r), and r is the number of atoms of [A B] smaller than the subproblem's atoms.

We now generalize the division of A from two to (n/lg n) subproblems. The sequence A will be divided into subsequences "group.k.(lg n).A" each of (lg n) atoms. Let **corres** define a subsequence of B that corresponds to one **group** of A:

$$\text{corres.k.A.B} \approx [\,\text{lcorr} \le i < \text{rcorr: B.}i\,],$$

$$\text{lcorr} \approx \textbf{if}\ \text{k}=0 \rightarrow 0\ \square\ \text{k}>0 \rightarrow \text{place.(k}*\text{lg n).A.B}\ \textbf{fi},$$

$$\text{rcorr} \approx \textbf{if}\ \text{k} \ge (\text{n/lg n}) \rightarrow \text{n}\ \square\ \text{k} < (\text{n/lg n}) \rightarrow \text{place.((k+1)}*\text{lg n).A.B}\ \textbf{fi}.$$

Elements of "corres.k.A.B" are less than "A.((k+1)*lg n)" and larger than "A.((k*lg n)-1)." Then **merge** is the concatenation of (n/lg n) **merge** subproblems:

$$\text{merge.k.C.A.B} = [\ 0 \le j < (\text{n/lg n}):\ \text{merge.q.C.(group.}j\text{.(lg n).A).(corres.}j\text{.A.B)}\ ],$$

$$\text{q} \approx \text{k} + j*(\text{lg n}) + \text{place.(}j*(\text{lg n})\text{).A.B}$$

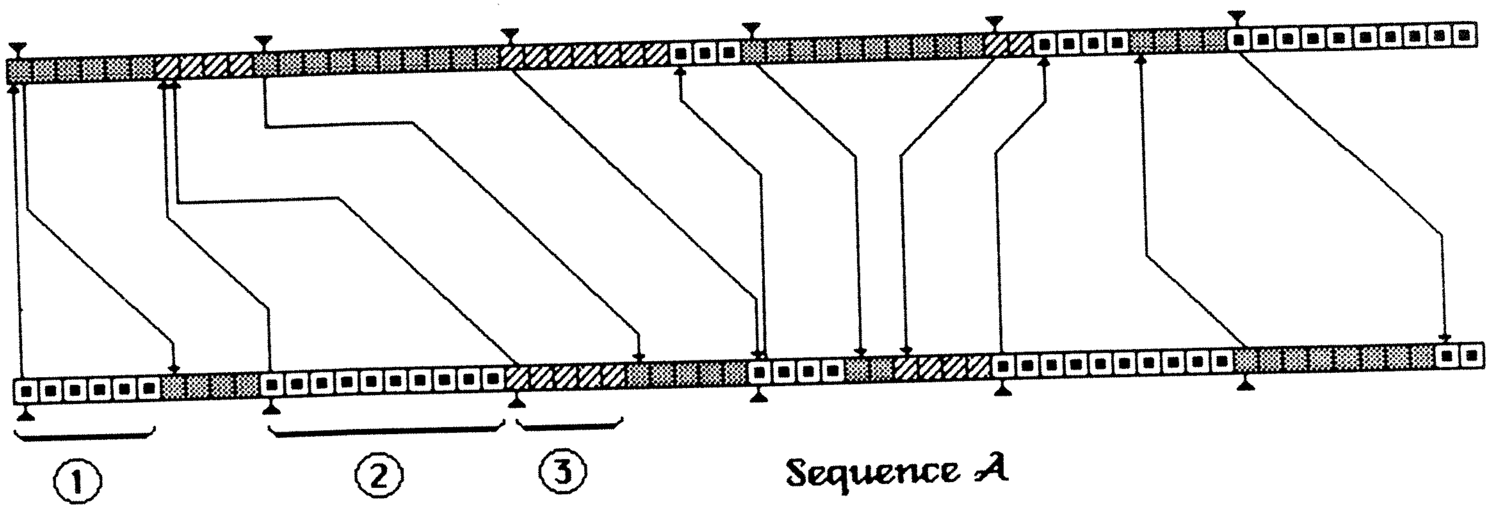
This decomposition is correct because the three obligations for subproblems are fulfilled. There is one scenario in which the formulation above can be computed in NC^1(n/lg n). If the size of each **corres** sequence is O(lg n), then each subproblem is NC^1(1) by a sequential **merge** computation. The computation of **place** is also NC^1(1), thus all subproblems may be computed in O(lg n) time using O(n/lg n) processors.

The remainder of the discussion is aimed at insuring that each subproblem merges sequences of size O(lg n). To this point, we have only considered dividing B with respect to some atoms of A. If we apply the same procedure with roles of A and B interchanged, then we divide B into regular, O(lg n) size intervals while A is divided into irregular sizes. If both sequences A and B are partitioned in a way that accomodates both types of division, regular and irregular, then each partition will contain at most (lg n) atoms.

Figure 1 is an illustration of our plan. In the figure, atoms A.k and B.k, for k a multiple of (lg n), are marked with triangles. We call these atoms *posts.* Arrows are drawn from the posts to where they fit in the opposite sequence: For instance from A.k to B.(place.k.A.B). The arrows are drawn from posts to *post counterparts.* By ascendency properties, these arrows never cross. The subproblems are defined by a pair of arrows and similarly shaded boxes; there are twelve subproblems in the figure. In the illustration, A.0 is smaller than B.0; the first subsequence of A, labeled (1), merges with an empty counterpart from B. (1) terminates before "A.(place.0.B.A)," which is B's first post counterpart. The subsequence that begins with A.(lg n), labeled (2), also has a null counterpart, this time because of a conflict with subsequence (3). Subsequence (3) terminates where B's second post has its counterpart.

Please see insert following page.

Figure 1. Subproblems for merge.


Sequence B
1
2
3
Sequence A

1.5.6.2 **Merge** is in NC^1(n/lg n).

Demonstration. Let **slots** be the function to compute **place** for all posts:

$$\text{slots.A.B} \approx [\ 0 \leq i < \text{n/lg n: place.}(i*\text{lg n}).\text{A.B}\ ]\ .$$

Computing **slots** is a problem in NC^1(n/lg n). Observe that **slots** is an ascending sequence, though not strictly ascending.

Let **uplim** define the upper limit for one of the sequences in the **merge** subproblem:

$$\text{uplim.j.B.A} \approx \textbf{if } x \geq n/(\text{lg n}) \rightarrow j+\text{lg n} \ [\!]\ x < n/(\text{lg n}) \rightarrow \text{min.}(j+\text{lg n}).((\text{slots.A.B}).x) \textbf{ fi},$$

$$x \approx \text{reduce.plus.}\ [\ 0 \leq i < n/(\text{lg n}):$$

$$\textbf{if } j > (\text{slots.A.B}).i \rightarrow 1 \ [\!]\ j \leq (\text{slots.A.B}).i \rightarrow 0 \textbf{ fi}\ ]\ .$$

The value x represents the number of A's post counterparts that are smaller than B.j. If B.j is smaller than some post counterpart of A, then "(slots.A.B).x" is the smallest post counterpart of A that is larger than B.j. The computation of x is in NC^1(1) by a binary search: Although **slots** is not strictly ascending, the binary search is applicable in time O(lg(n/lg n)).

One half of the merge subproblem is given by the sequence **primepart**:

$$\text{primepart.j.B.A} \approx [\ j \leq i < \text{uplim.j.B.A: B.}i\ ]\ .$$

This sequence begins at a post in B and continues until (1) the next post B or (2) a post counterpart from A. The reader may verify that the definition of **uplim** insures that **primepart** is neither an empty sequence nor a sequence with more that (lg n) atoms. To define the sequence that corresponds to **primepart** we first define its upper limit, **scancross**:

scancross.k.m.B.A ≈ **if**

(m mod (lg n))=0→ m ▯

(m mod (lg n))≠0→ **if**

(k+lg n)≥n→ scancross.k.(m+1).B.A ▯

(k+lg n)<n→ **if**

m=place.(k+lg n).B.A→ m ▯

m≠place.(k+lg n).B.A→ scancross.k.(m+1).B.A **fi**

fi

fi.

Despite its complicated appearance, **scancross** is a simple search with some endpoint conditions. Intially, **scancross** is invoked "scancross.k.k.B.A." Starting at A.k (which will turn out to be a post counterpart of B), **scancross** searches sequentially through A for (1) a post in A, (2) the end of A, or (3) the next post counterpart of B. This search is a search of at most (lg n) atoms, so its classification is $NC^1(1)$.

Now **secpart**, which is the sequence corresponding to **primepart** is defined:

secpart.j.B.A ≈ [place.j.B.A≤*i*<scancross.j.j.B.A: A.*i*] .

From the definition of **scancross**, the sequence **secpart** may be a null sequence, but may not have more than (lg n) atoms.

Merge is composed of the 2n/(lg n) subproblems:

merge.k.C.A.B = [[0≤*j*<n/(lg n):

merge.(k+v).C.(primepart.(*j*∗(lg n)).B.A).(secpart.(*j*∗(lg n)).B.A)]

[0≤*j*<n/(lg n):

$$\text{merge.(k+w).C.(primepart.}(j*(\lg n))\text{.A.B).(secpart.}(j*(\lg n))\text{.A.B)]]},$$

$$v \approx (j*(\lg n)) + \text{place.}(j*(\lg n))\text{.B.A}$$

$$w \approx (j*(\lg n)) + \text{place.}(j*(\lg n))\text{.A.B.}$$

The functions v and w have been designed to satisfy correctness obligation (iii), which places the output of each subproblem in the proper location. The definitions of **primepart** and **secpart** satisfy obligations (i) and (ii).

Each subproblem is defined by **primepart** and **secpart** sequences, the limits of which can be calculated in O(lg n) time with a single processor. It follows that all subproblem limits are found in NC^1(n/lg n). The **merge** subproblems combine inputs each of size O(lg n), so each subproblem is in NC^1(1) by a sequential merge. All subproblems are computed in NC^1(n/lg n). ∎

1.6 *Summary*

We have demonstrated efficient parallel paradigms for classes of linear problems. The example applications suggest the generality of the paradigms. Other evidence is found in the study of programming languages. For example, a recent release of the APL programming language [14] contains functionals that correspond to **apply**, **reduce** and **scan**. Programmers are encouraged to use these functionals because they are primitive features of the language.

Most of the efficient parallel algorithms of this paper use bounded parallel sharing of memory. When the sharing is bounded, meaning that only a fixed number of processors simul-

taneously read a common unit of memory, a simpler model than the PRAM can be used by interleaving the memory references. It may be possible for our results to be translated to permutation networks or distributed systems. Only in the **merge** problem, which specifies 2n/(lg n) parallel binary searches is the sharing unbounded.

In view of the results of the previous sections, an interesting question is:

$$P^{1} \cap NC^{1} = NC^{1}(n/\lg n)\ ?$$

All examples in this paper satisfy the conjecture. An open question, noted in [13], is the problem of computing the median of n numbers. The classifications P^{1} and NC^{1} are known, but there is no apparent $NC^{1}(n/\lg n)$ algorithm.

1.7 *References*

[1] S. A. Cook, The Classification of Problems which have Fast Parallel Algorithms, Technical Report No. 164/83, University of Toronto.

[3] D. S. Hirschberg, A. K. Chandra and D. V. Sarwate, Computing Connected Components on Parallel Computers, *Comm. ACM 22* 8(Aug 79), pp. 461-464.

[4] F. Y. Chin, J. Lam, and I. Chen, Efficient Parallel Algorithms for Some Graph Problems, *Comm. ACM 25* 9(Sep 82), pp. 659-665.

[6] D. S. Johnson, The NP-Completeness Column: An Ongoing Guide, *J. Algorithms 4* 1983, pp. 189-203.

[7] R. E. Tarjan, Complexity of Combinatorial Algorithms, *SIAM Review 20* 3(Jul 78), pp. 457-491.

[8] A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms,* 1974, Addison-Wesley, Reading Massachusetts.

[9] C. L. Liu, *Elements of Discrete Mathematics,* 1977 McGraw-Hill, New York.

[10] E. W. Dijkstra, *A Discipline of Programming,* Prentice Hall, New York.

[11] J. T. Schwartz, Ultracomputers, *ACM Transactions on Programming Languages and Systems 2* 4(Oct 80), pp. 484-521.

[12] K. E. Batcher, Sorting networks and their applications, *Proc. 1968 JCC,* AFIPS Press, Arlington Virginia, pp. 307-314.

[13] S. G. Akl, An optimal algorithm for parallel selection, *Information Processing Letters 19* (1984), pp. 47-50.

[14] *APL2 Programming: Language Reference,* 1984, IBM publication SH20-9227-0.