SPECIAL ISSUE

Articles

DATA PARALLEL ALGORITHMS

Parallel computers with tens of thousands of processors are typically programmed in a data parallel style, as opposed to the control parallel style used in multiprocessing. The success of data parallel algorithms—even on problems that at first glance seem inherently serial—suggests that this style of programming has much wider applicability than was previously thought.

W. DANIEL HILLIS and GUY L. STEELE, JR.

In this article we describe a series of algorithms appropriate for fine-grained parallel computers with general communications. We call these algorithms *data parallel* algorithms because their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control. The intent is not so much to present new algorithms (most have been described earlier in other contexts), but rather to demonstrate a style of programming that is appropriate for a machine with tens of thousands or even millions of processors. The algorithms described all use $O(N)$ processors to solve problems of size N, typically in $O(\log N)$ time. Some of the examples solve problems that at first sight seem inherently serial, such as parsing strings and finding the end of a linked list. In many cases, we include actual run times measured on the 65,536-processor Connection Machine® system. A key feature of this hardware is a general-purpose communications network connecting the processors that frees the programmer from detailed concerns of the mapping between data and hardware.

MODEL OF THE MACHINE

Our model of a fine-grained parallel machine with general-purpose communication is based on the Connection Machine system [14]. Current Connection Machine systems have either 16,384 or 65,536 processors, each with 4,096 bits of memory. There is nothing special about these particular numbers, other than being powers of two, but they are indicative of the scale of the machine. (The model is more sensible with thousands rather than tens of processors.) The system has two parts: a front-end computer of the usual von Neumann style, and an array of Connection Machine processors. Each processor in the array has a small amount of local memory, and to the front end, the processor array looks like a memory. A typical front-end processor is a VAX® or a Symbolics 3600®.

The processor array is connected to the memory bus of the front end so that the local processor memories can be random accessed directly by the front end, one word at a time, just as if it were any other memory. This section of the memory on the front end, however, is "smart" in the sense that the front end can issue special commands that cause many parts of the memory to be operated upon simultaneously, or cause data to move around within the memory. The processor array therefore effectively extends the instruction set of the front-end processor to include instructions that operate on large

Connection Machine is a registered trademark of Thinking Machines Corporation.

VAX is a trademark of Digital Equipment Corporation.

Symbolics 3600 is a trademark of Symbolics, Inc.

amounts of data simultaneously. In this way, the processor array serves a function similar to a floating-point accelerator unit, except that it accelerates general parallel computation and not just floating-point arithmetic.

The control structure of a program running on a Connection Machine system is executed by the front end in the usual way. An important practical benefit of this approach is that the program is developed and executed within the already-familiar programming environment of the front end. The program can perform computations in the usual serial way on the front end and also issue commands to the processor array.

The processor array executes commands in SIMD fashion. There is a single instruction stream coming from the front end; these instructions act on multiple data items, on the order of one (or a few) per processor. Most instructions are executed conditionally: That is, each processor has state bits that determine which instructions the processor will execute. A processor whose state bit is set is said to be *selected*. The state bit is called the *context flag* because the set of selected processors is often referred to as the *context* within which instructions are executed. For example, the front end might arrange for all odd-numbered processors to have their context flags set, and even-numbered processors to have their context flags cleared; issuing an **ADD** instruction would then cause each of the selected processors (the odd-numbered ones) to add one word of local memory into another word. The deselected (even-numbered) processors would do nothing, and their local memories would remain unchanged.

Contexts may be saved in memory and later restored, with each processor saving or restoring its own bit in parallel with the others. There are a few instructions that are unconditional: They are executed by every processor regardless of the value of the context flag. Such instructions are required for saving and restoring contexts.

A context, or a set of values for all the context flags, represents a set: namely, a set of selected processors. Forming the intersection, union, or complement of such sets is simple and fast; it requires only a one-bit logical **AND, OR,** or **NOT** operation issued to the processors. Locally viewed, each processor performs just one logical operation on one or two single-bit operands; viewed globally, an operation on sets is performed. (On the current Connection Machine hardware, such an operation takes about a microsecond.)

The processors can individually perform all the usual operations on logical, integer, and floating-point operands: add, subtract, multiply, divide, compare, max, min, not, and, or, exclusive or, shift, square root, and so on. In addition, single values computed in the front end can be broadcast from the front end to all processors at once (essentially by including them as immediate data in the instruction stream).

A number of other computing systems have been constructed with the characteristics we have already described, namely, a large number of parallel processors, each of which has local memory and the ability to execute instructions of more or less the usual sort, as broadcast from a master controller. These include ILLIAC IV [8], the Goodyear MPP [3], the Non-Von [23]; and the ICL DAP [12]; among others [13]. There are two additional characteristics of the Connection Machine programming model, however, which distinguish it from these other systems: *general, pointer-based communication,* and *virtual processors.*

Previous parallel computing systems of this fine-grained SIMD style have restricted interprocessor communication to certain patterns wired into the hardware; typically this pattern is a two-dimensional rectangular grid, or a tree. The Connection Machine model allows any processor to communicate directly with any other processor in unit time, while other processors also communicate concurrently. Communication is implemented via a **SEND** instruction.

Within each processor, the **SEND** instruction takes two operands: One addresses—within the processor—the field that contains the data to be sent; the other addresses a processor pointer (i.e., the number of the processor to which the datum is to be sent and the destination field within that processor, into which the data will be placed). The communications system is very much like a postal system, where you can send a message to anyone else directly, provided you know the address, and where many letters can be routed at the same time. The **SEND** instruction can also be viewed as a parallel "store indirect" instruction that allows each processor to store anywhere in the entire memory, not just in its own local memory.

The **SEND** instruction can also take one additional operand that specifies what happens if two or more messages are sent to the same destination. The options are to deliver to the destination the sum, maximum, minimum, bitwise **AND**, or bitwise **OR** of the messages; to deliver one message and discard all others; or to produce an error.

From a global point of view, the **SEND** instruction performs something like an arbitrary permutation on an array of items, although it is actually more

general than a permutation because more than one item may be sent to the same destination. The pattern is not wired into the hardware, but is encoded as an array of pointers, and is completely under software control; the same encoding, once constructed, can be used over and over again to transfer data repeatedly in the same pattern. To implement a regular communication pattern, such as a two-dimensional grid, the pointers are typically computed when needed rather than stored.

The Connection Machine programming model is carefully abstracted from the details of the hardware that supports it, and, in particular, the number and size of its hardware processors. Programs are described in terms of virtual processors. In actual implementations, hardware processors are multiplexed as necessary to support this abstraction; indeed, the abstraction is supported at a very low level by a microcoded controller interposed between the front end and the processor array, so that the front end always deals in virtual processors.

The benefits of the virtual processor abstraction are twofold. The first is that the same program can be run unchanged on different sizes of the Connection Machine system, notwithstanding the linear trade-off between the number of hardware processors and execution time. For example, a program that requires 2^{16} virtual processors can run at top speed on a system with the same number of hardware processors, but it can also be executed on one-fourth that amount of hardware (2^{14} processors) at one-fourth the speed, provided it can fit into one-fourth the amount of memory as well.

The second benefit is that for many purposes the number of processors may be regarded as expandable rather than fixed, so that it becomes natural to write programs using the Lisp, Pascal, or C style of storage allocation rather than the Fortran style. By this we mean that there is a procedure one can call to allocate a "fresh" processor as if from thin air while the program is running. In Fortran, all storage is preallocated before program execution begins, whereas Lisp has the **cons** operation to allocate a new list cell (as well as other operations for constructing other objects); Pascal has the **new** operation; and C has the **malloc** function. In each case, a new object is allocated and a pointer to this new object is returned. Of course, in the underlying implementation, the address space (physical or virtual) is actually a fixed resource pool from which all such requests are satisfied, but the point is that the language supports the abstraction of newly created storage. In the Connection Machine model, one may similarly allocate fresh storage using the operation **processor-cons**; the difference is that the newly allocated storage comes with its own processor attached.

In the ensuing discussion, we shall assume that the size and number of processors are sufficient to allocate one processor for each data element in the problem being solved. This allows us to adopt a model of the machine in which the following are counted as unit-time operations:

- any conventional word-at-a-time operation;
- any such operation applied to all the data elements concurrently, or to some selected subset;
- any communications step that involves the broadcast of information to all data elements;
- any communications step that involves no more than a single message transmission from each data element.

For purposes of analysis, it is also often useful to treat **processor-cons** as a unit-time operation, although it may be implemented in terms of more primitive operations, as described in more detail on page 1176.

EXAMPLES OF PARALLEL PROGRAMMING

To show some of the possibilities of data-parallel programming, we present here several algorithms currently in use on the Connection Machine system. Most of these algorithms are not new: Some of the ideas represented here appear in the languages APL [11, 15] and FP [1], while others are based on algorithms designed for other parallel machines, in particular, the Ultracomputer [22], and still others have been developed by our coworkers on the Connection Machine [4, 5, 7, 9, 10, 19].

Beginning with some very simple examples to familiarize the reader with the model and the notation, we then proceed to more elaborate and less obvious examples.

Sum of an Array of Numbers

The sum of n numbers can be computed in time $O(\log n)$ by organizing the addends at the leaves of a binary tree and performing the sums at each level of the tree in parallel. There are several ways of organizing an array into a binary tree. Figure 1 illustrates one such method on an array of 16 elements named x_0 through x_{15}. In this algorithm, for purposes of simplicity, the number of elements to be summed is assumed to be an integral power of two. There are as many processors as elements, and the statement **for all** k **in parallel do** s **od** causes all processors to execute the same statement s in synchrony, but the variable k has a different value for each processor,

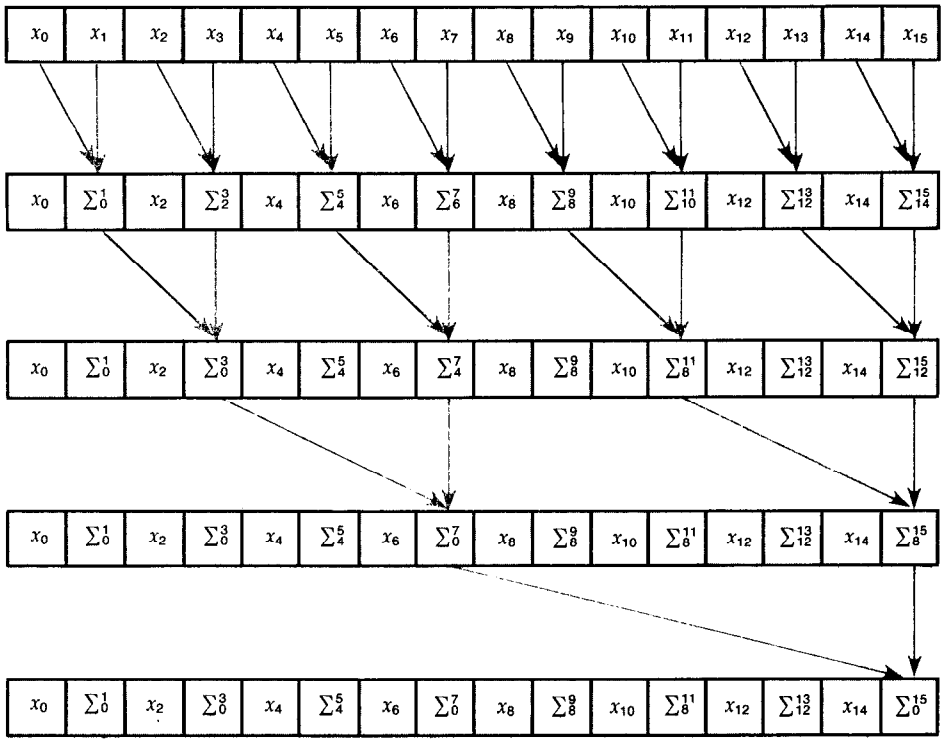


FIGURE 1. Computing the Sum of an Array of 16 Elements

namely, the index of that processor within the array.

```
for j := 1 to log2 n do
  for all k in parallel do
    if ((k + 1) mod 2^j) = 0 then
      x[k] := x[k − 2^(j−1)] + x[k]
    fi
  od
od
```

At the end of the process, x_{n-1} contains the sum of the n elements. On the Connection Machine, an optimized version of this algorithm for 65,536 elements takes about 200 microseconds.

All Partial Sums of an Array

A frequently useful operation is computing all partial sums of an array of numbers. In APL, this computation is called a plus-scan; in other contexts, it is called the "sum-prefix" operation because it computes sums over all prefixes of the array. For example, if you put into an array your initial checkbook balance, followed by the amounts of the checks you have written as negative numbers and deposits as positive numbers, then computing the partial sums produces all the intermediate and final balances.

It might seem that computing such partial sums is an inherently serial process, because one must add up the first k elements before adding in element $k + 1$. Indeed, with only one processor, one might as well do it that way, but with many processors one can do better, essentially because in $\log n$ time with n processors one can do $n \log n$ individual additions; serialization is avoided by performing logically redundant additions.

Looking again at the simple summation algorithm given on the facing page, we see that most of the processors are idle most of the time: During iteration j, only $n/2^j$ processors are active, and, indeed, half of the processors are never used. However, by putting the idle processors to good use by allowing more processors to operate, the summation algorithm can compute all partial sums of the array in the same amount of time it took to compute the single sum. In defining $\sum_j^k$ to mean $\sum_{i=j}^k x_i$, note that $\sum_j^k + \sum_{k+1}^m = \sum_j^m$. The partial-sums algorithm replaces each x_k by $\sum_0^k$: that is, the sum of all elements preceding and including x_k. In Figure 2 (on the following page), this process is illustrated for an array of 16 elements.

```
for j := 1 to log2 n do
  for all k in parallel do
    if k ≥ 2^j then
      x[k] := x[k − 2^(j−1)] + x[k]
    fi
  od
od
```

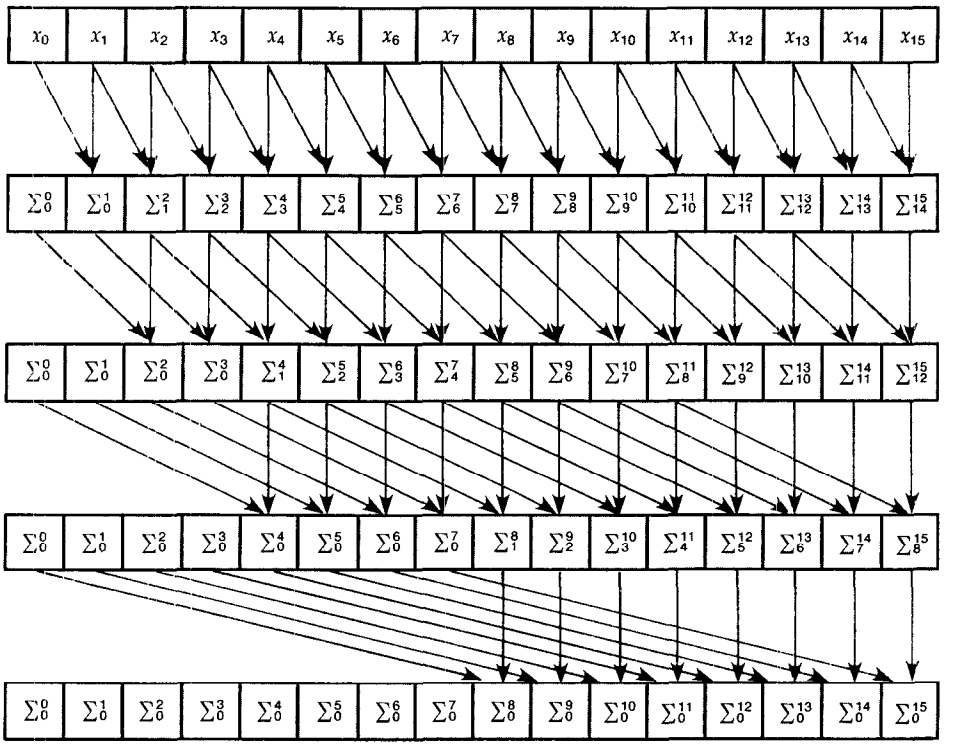


FIGURE 2. Computing Partial Sums of an Array of 16 Elements

The only difference between this algorithm and the earlier one is the test in the **if** statement in the partial-sums algorithm that determines whether a processor will perform the assignment. This algorithm keeps more processors active: During step j, $n - 2^{j-1}$ processors are in use; after step j, element number k has become $\sum_a^k$ where $a = \max(0, k - 2^j + 1)$.

This technique can be generalized from summation to any associative combining operation. Some obvious choices are product, maximum, minimum, and logical **AND**, **OR**, and **EXCLUSIVE OR**. Some programming languages provide such reduction and parallel-prefix operations on arrays as primitives. The current proposal for Fortran 8x, for example, provides reduction operations called **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **ANY**, and **ALL**. The corresponding reduction functions in APL are +/, ×/, ⌈/, ⌊/, ∨/, and ∧/; APL also provides other reduction operations and all the corresponding scan (prefix) operations. The combining functions for all these operations happen to be commutative as well, but the algorithm does not depend on commutativity. This was no accident; we took care to write

$$x[k] := x[k - 2^{j-1}] + x[k]$$

instead of the more usual

$$x[k] := x[k] + x[k - 2^{j-1}]$$

precisely in order to preserve the correctness of the algorithm when + is replaced by an associative but noncommutative operator. Under "Parsing a Regular Language" (facing page), we discuss the use of parallel-prefix computations with a noncommutative combining function for a nonnumerical application, specifically, the division of a character string into tokens. Another associative noncommutative operator of particular practical importance is matrix multiplication. We have found this technique useful in multiplying long chains of matrices.

Counting and Enumerating Active Processors

After some subset of the processors has been selected according to some condition (i.e., by using the result of a test to set the context flags), two operations are frequently useful: determining how many processors are active, and assigning a distinct integer to each processor. We call the first operation **count** and the second **enumerate**: Both are easily implemented in terms of summation and sum-prefix.

To **count** the active processors, we have every processor unconditionally examine its context flag and compute the integer 1 if the flag is set and 0 if it is clear. (Remember that an unconditional operation is performed by every processor regardless of whether or not its context flag is set.) We then perform an unconditional summation of these integer values.

To **enumerate** the active processors, we have every processor unconditionally compute a 1 or 0 in the same manner, but then we perform an unconditional sum-prefix calculation with the result that every processor receives a count of the number of active processors that precede it (including itself) in the ordering. We then revert to conditional operation; in effect, the selected processors have received distinct integers, and values computed for the deselected processors are henceforth simply ignored. Finally, it is technically convenient to have every selected processor subtract one from its value, so that the n selected processors will receive values from 0 to $n - 1$ rather than from 1 to n.

These operations each take about 200 microseconds on a Connection Machine of 65,536 elements. Because these operations are so fast, programming models of the Connection Machine have been suggested that treat counting and enumeration as unit-time operations [6, 7].

Radix Sort

Sorting is a communications-intensive operation. In parallel computers with fixed patterns of communication, the pattern of physical connections usually suggests the use of a particular sorting algorithm. For example, Batcher's bitonic sort [2] fits nicely on processors connected in a perfect shuffle pattern, bubble sorts [16] work well on one-dimensionally connected processing, and so on. In the Connection Machine model, the ability to access data in parallel in any pattern often eliminates the need to sort data. When it is necessary to sort, however, the generality of communications provides an embarrassment of riches.

Upon implementing several sorting algorithms on the Connection Machine and comparing them for speed, we found that, for the current hardware implementation, Batcher's method has good performance for large sort keys, whereas for small sort keys a version of radix sort is usually faster. (The break-even point is for keys about 25 to 32 bits in length. With either algorithm, sorting 65, 536 32-bit numbers on the Connection Machine takes about 30 milliseconds.)

To illustrate the use of **count** and **enumerate**, we present here the radix sort algorithm. In the interest of simplicity, we will assume that all processors (n) are active, that sort keys are unsigned integers, and that *maxint* is the value of the largest representable key value.

```
for j := 1 to 1 + ⌊log2 maxint⌋ do
  for all k in parallel do
    if (x[k] mod 2^j) < 2^(j-1) then
      comment The bit with weight 2^(j-1) is zero.
      tnemmoc
      y[k] := enumerate
      c := count
    fi
    if (x[k] mod 2^j) ≥ 2^(j-1) then
      comment The bit with weight 2^(j-1) is one.
      tnemmoc
      y[k] := enumerate + c
    fi
    x[y[k]] := x[k]
  od
od
```

At this point, an explanation of a fine point concerning the intended semantics of our algorithmic notation is in order. An **if** statement that is executed for all processors is always assumed to execute its **then** part, even if no processors are selected, because some front-end computations (such as **count**) might be included. When the **then** part is executed, the active processors are those previously active processors that computed **true** for the condition.

The radix sort requires a logarithmic number of passes, where each pass essentially examines one bit of each key. All keys that have a 0 in that bit are counted (call the count c) and then enumerated in order to assign them distinct integers y_k ranging from 0 to $c - 1$. All keys that have a 1 in that bit are then enumerated, and c is added to the result, thereby assigning these keys distinct integers y_k ranging from c to $n - 1$. The values y_k are then used to permute the keys so that all keys with a 0 bit precede all keys with a 1 bit. (This is the step that takes particular advantage of general communication.) This permutation is stable: The order of any two keys that both have 0 bits or both 1 bits is preserved. This stability property is important because the keys are sorted by least significant bit first and most significant bit last.

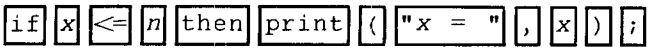
Parsing a Regular Language

To illustrate the use of a parallel-prefix computation in a nonnumerical application, consider the problem of parsing a regular language. For a concrete practical instance, let us consider the problem of breaking

up a long string of characters into tokens, which is usually the first thing a compiler does when processing a program. A string of characters such as

```
if x <=n then print("x = ", x);
```

must be broken up into the following tokens, with redundant white space eliminated:



This process is sometimes called *lexing a string.*

Any regular language of this type can be parsed by a finite-state automaton that begins in a certain state and makes a transition from one state to another (possibly the same one) as each character is read. Such an automaton can be represented as a two-dimensional array that maps the old state and the character read to the new state. Some of the states correspond to the start of a token; if the automaton is in one of those states just after reading a character, then that character is the first character of a token. Some characters may not be part of any token; White-space characters, for example, are typically not part of a token unless they occur within a string; such delimiting characters may also be identified by the automaton state just after the character is read. To divide a string up into tokens, then, means merely determining the state of the automaton after each character has been processed.

Table I shows the automaton array for a simple language in which a token may be one of three things: a sequence of alphabetic characters, a string surrounded by double quotes (where an embedded double quote is represented by two consecutive double quotes), or any of +, −, *, =, <, >, <=, and >=. Spaces and newlines delimit tokens, but are not part of any token except quoted strings. The automaton has nine states: *N* is the initial state; *A* is the start of an alphabetic token; Z is the continuation of an alphabetic token; * is a single-special-character token; < is a < or > character; = is an = that follows a < or > character (an = that does not follow < or > will produce state *); *Q* is the double quote that starts a string; *S* is a character within a string; and *E* is the double quote that ends a string, or the first of two that indicate an embedded double quote. The states *A*, *, <, and *Q* indicate that the character just read is the first character of a token.

Although, like the computation of partial sums, this may appear at first glance to be an inherently serial process, it too can be put into the form of a parallel-prefix computation. Rather than regarding the lexing automaton as a monolithic process, let us regard the individual characters of the string as unary functions that map an automaton state onto another state. By indicating the application of the character Y to state *N* as *NY*, we may then write *NY* = *A*. By extension, it is also possible to regard a string as a function that maps a state *p* to another state *q*; *q* is the state you end up in if you start the automaton in state *p* and then let the automaton read the entire string one character at a time. The result of applying the string Y″+ to the state Z may be written as ZY″+ = ((ZY)″)+ = (Z″)+ = *Q*+ = *S*. It is not too hard to see that the function corresponding to a string is simply the composition of the functions for the individual characters.

A function from a state to a state can be represented as a one-dimensional array indexed by states whose elements are states. The columns of the array in Table I are in fact exactly such representations for the functions for individual characters. Composing the columns for two characters or strings to produce a new column for the concatenation of the strings is fairly straightforward: You simply replace every entry of one column with the result of using that entry to index into the other column.

Since this composition operation is associative, we may compute the automaton state after every character in a string as follows:

1. Replace every character in the string with the array representation of its state-to-state function.
2. Perform a parallel-prefix operation. The combining function is the composition of arrays as described above. The net effect is that, after this step, every character *c* of the original string has been replaced by an array representing the state-to-state function for that prefix of the original string that ends at (and includes) *c*.
3. Use the initial automaton state (*N* in our example) to index into all these arrays. Now every character has been replaced by the state the automaton would have after that character.

If we implement this algorithm on a Connection Machine system and allot one processor per character, the first and third steps will take constant time, and the second step will take time logarithmic in the length of the string. Naturally, this algorithm performs much more computation per character than the straightforward serial algorithm using the two-dimensional array, but, for sufficiently large amounts of text, the parallel algorithm will be faster because its time complexity is logarithmic instead of linear. An implementation of this algorithm in Connection Machine Lisp can be found in [24].

PARALLEL PROCESSING OF POINTERS

Processor-cons

To illustrate pointer manipulation algorithms, we will consider the implementation of the **processor-cons** primitive, which allows a set of processors to establish pointers to a set of new processors allocated from free storage. In a serial computer, the equivalent problem is usually solved by keeping the free storage in an ordered list and allocating new storage from the beginning of the list. In the Connection Machine, this would not suffice since we wish to allocate many elements concurrently. Instead, the **processor-cons** primitive is implemented in terms of **enumerate** by using a rendezvous technique: Two sets of m processors are brought into one-to-one communication by using the processors numbered 0 through $m - 1$ as rendezvous points.

Assuming that every processor has a Boolean variable called *free*, $free_k$ becomes **true** if processor k is available for allocation and **false** otherwise. Every selected processor is to receive, in a variable called *new-processor*, the number of a distinct free processor. Free processors that are so allocated have their *free* bits reset to **false** in the process. If there are fewer free processors than selected processors, then as many requests are satisfied as possible, and some selected processors will have their *new-processor* variables set to *null* instead of the number of a free processor, as shown below.

```
for all k in parallel do
  required := count
  unconditionally
    if free[k] then
      available := count
      free-processor[k] := enumerate
      if free-processor[k] < required then
        free[k] := false
      fi
      rendezvous[free-processor[k]] := k
    requestor[k] := enumerate
    fi
  yllanoitidnocnu
  if requestor[k] < available then
    new-processor := rendezvous[requestor[k]]
  else
    new-processor := null
  fi
od
```

In this way, the total number of processors is managed as a finite resource, but with an interface that presents the illusion of creating new processors on demand. (Of course, we have not indicated how processors are returned to the pool of free processors. Some technique such as reference counting or garbage collection must also be designed and coded.) Other algorithms for **processor-cons** are described in [9, 10].

TABLE I. A Finite-State Automaton for Recognizing Tokens

Old State	Character Read													
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E	S	S
S	S	S	...	S	S	S	S	S	S	S	S	E	S	S
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

Parallel Combinator Reduction

A topic of much current interest in the area of functional programming is parallel combinator reduction [25]. It is also particularly interesting in this context because it shows how data parallel algorithms can be used to simulate control parallelism, or, equivalently, how SIMD machines with general communication can simulate MIMD machines.

Combinators are a way of encoding an applicative language. Their appeal lies in the fact that a program can be executed simply by performing successive local transformations on a tree structure, moreover, it is possible to perform many independent transformations simultaneously in the same tree. A combinator tree is made up of pairs, where each of the *left* and *right* components of a pair may point to another pair or else be an atom, the name of a combinator. Standard names for combinators include S, K, I, B, and C. Figure 3 (next page) shows one possible set of four transformations that suffices for program interpretation. When a subtree is transformed, the root pair of the subtree is used as the root pair of the result (by altering its components), but it is not permissible to alter any of the other pairs involved; therefore, the transformation involving the S combinator requires the allocation of fresh pairs. For our purposes, we ignore the semantics of the combinators and simply observe that such graph transformations can easily be carried out in parallel by a Connection Machine system by letting each processor contain one pair, and using **processor-cons** to allocate new pair-processors as needed.

```
while want or need to reduce some more do
  for all k in parallel do
    lf := left[k]
    if pair(lf) then
      if left[lf] = 'K' then
        left[k] := 'I'
      fi
      if left[lf] = 'I' then
        left[k] := right[lf]
      fi
      if pair(left[lf]) and left[left[lf]] = 'S' then
        p := processor-cons
        q := processor-cons
        if p ≠ null and q ≠ null then
          left[p] := right[left[lf]]
          right[p] := right[lf]
          left[q] := right[left[lf]]
          right[q] := right[k]
          left[k] := p
          right[k] := q
        fi
      fi
    fi
    rt := right[k]
    if pair(rt) and left[rt] = 'I' then
      right[k] := right[rt]
    fi
  od
  possibly perform garbage collection
od
```

It is easy to write such parallel code as a Connection Machine program. However, there are some difficult resource-management issues that have been glossed over, such as when a garbage collection should occur; whether, at any given time, one should prefer to reduce S combinators or K combinators; and, given that S combinators are to be reduced, which ones should be given preference. (The issues are that there may be more of one kind of combinator than the other at any instant. One idea is to process whichever kind is preponderant, but S combinators consume pairs and K combinators *may* release pairs, so the best processing strategy may need to take the number of free processors available for allocation into account. Furthermore, if there are not enough free processors to satisfy all outstanding S combinators at once, then the computation may diverge—even if normal-order serial reduction would converge—if preference is consistently given to the wrong ones.)

Finding the End of a Linked List

When we first began to work with pointer structures in the Connection Machine model, we believed that balanced trees would be important because information can be propagated from the root of a tree to its leaves—or from the leaves to the root—by parallel methods that take time logarithmic in the number of leaves. This was correct. However, our intuition also told us that linear linked lists would be useless. We could understand how to process an array in logarithmic time, because one can use address arithmetic to compute the number of any processor and then communicate with it directly, but it seemed to us that a linked list must be processed serially because in general one cannot compute the address of the ninth processor in a list without following all eight of the intervening pointers.

As is so often true in computer science, intuition was misleading: Essentially, we overlooked the power of having many processors working on the problem at once. It is true that one must follow all eight pointers, but by using many processors one can still achieve this in time logarithmic in the number of pointers to be traversed. Although we found this algorithm surprising, it had been discovered in other contexts several times before (e.g., see chapter 9 of [20]).

As a simple example, consider finding the last cell of a linearly linked list. Imagine each cell to have a *next* pointer that points to the next cell in the list,

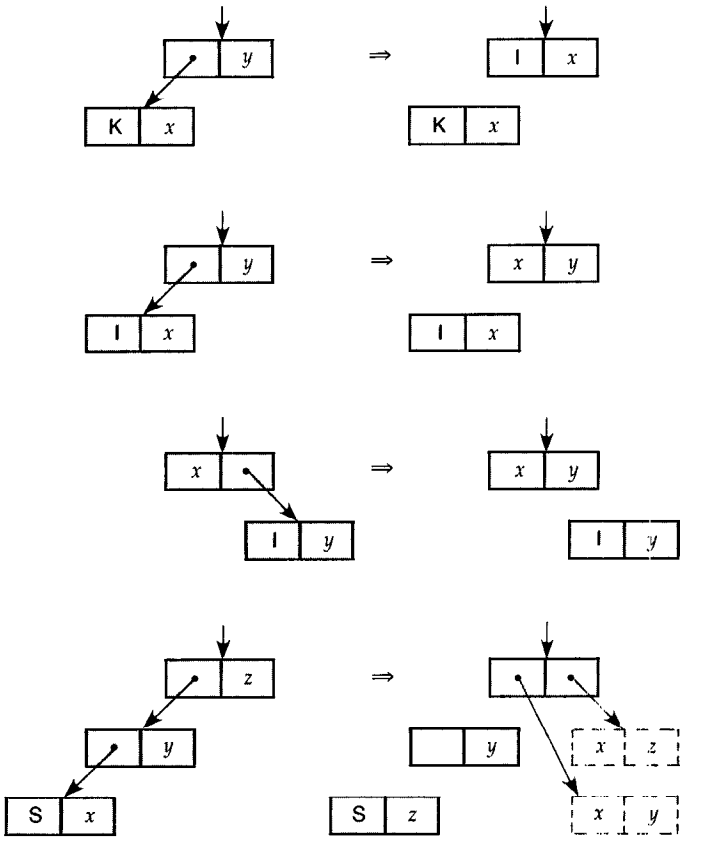


FIGURE 3. Patterns of Combinator Reduction

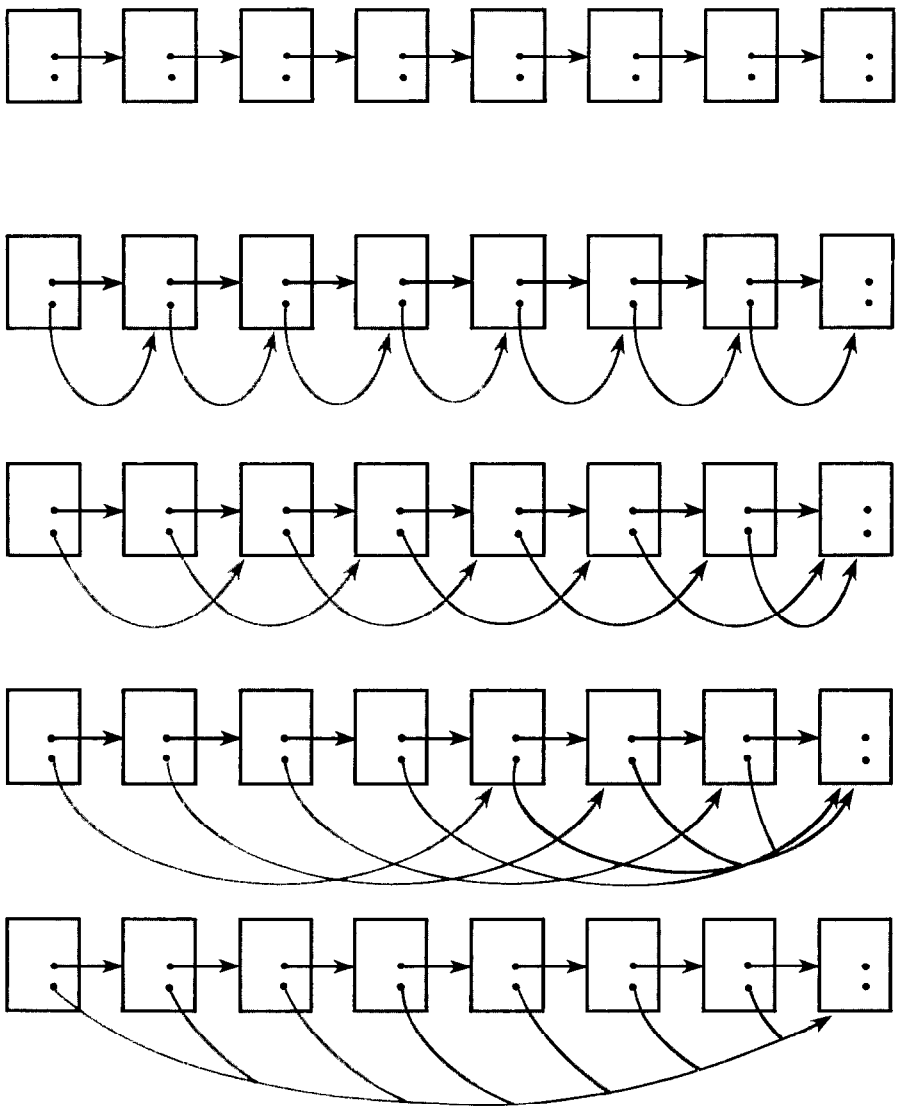


FIGURE 4. Finding the End of a Serially Linked List

while the last cell has the special value *null* in its *next* component. To accommodate other information as well, we will assume that in each cell there is another pointer component called *chum* that may be used for temporary purposes.

The basic idea is as follows: Each processor sets its *chum* component equal to a copy of its *next* component, so *chum* points to the next cell in the list. Each processor then repeatedly replaces its *chum* by its *chum's chum*. However, if its *chum* is *null*, then it remains *null*.) Initially, the *chum* of a processor is the next cell in the list; after the first step, its *chum* is the second cell following; after the second step, its *chum* is the fourth cell following; after the third step, its *chum* is the eighth cell following; and so on.

To ensure that the first cell of a list finds the last cell of a list, we carry out this procedure with the modification that a processor does not change its *chum* if its *chum's chum* is *null*, as shown below. The process is illustrated graphically in Figure 4.

```
for all k in parallel do
  chum[k] := next[k]
  while chum[k] ≠ null and chum[chum[k]] ≠ null do
    chum[k] := chum[chum[k]]
  od
od
```

The meaning of the **while** loop is that at each iteration a processor becomes deselected if it computes **false** for the test expression; the loop terminates when all processors have become deselected (whereupon the original context, as of the beginning of the loop, is restored). When this process terminates, *every* cell of the list except the last will have the last cell as its *chum*. If there are many lists in the machine, they can all be processed simultaneously,

and the total processing time will be proportional to the logarithm of the length of the longest such list.

All Partial Sums of a Linked List

The partial sums of a linked list may be computed by the same technique

```
for all k in parallel do
  chum[k] := next[k]
  while chum[k] ≠ null do
    value[chum[k]] := value[k] + value[chum[k]]
    chum[k] := chum[chum[k]]
  od
od
```

as illustrated in Figure 5. Comparing Figure 5 to Figure 2 (computing partial sums), we see that the same patterns of pointers among elements are constructed on the fly by using address arithmetic in the case of an array and by following pointer chains in the case of a linked list. An advantage of the linked-list representation is that it can simultaneously process many linked lists of different lengths without any change to the code.

Matching Up Elements of Two Linked Lists

An even more exotic effect is obtained by the following algorithm, which matches up corresponding elements in two linked lists. If we will call corresponding elements of a list "friends", this algorithm assigns to each list cell a pointer to its friend in the other list; of course, the result is obtained in logarithmic time.

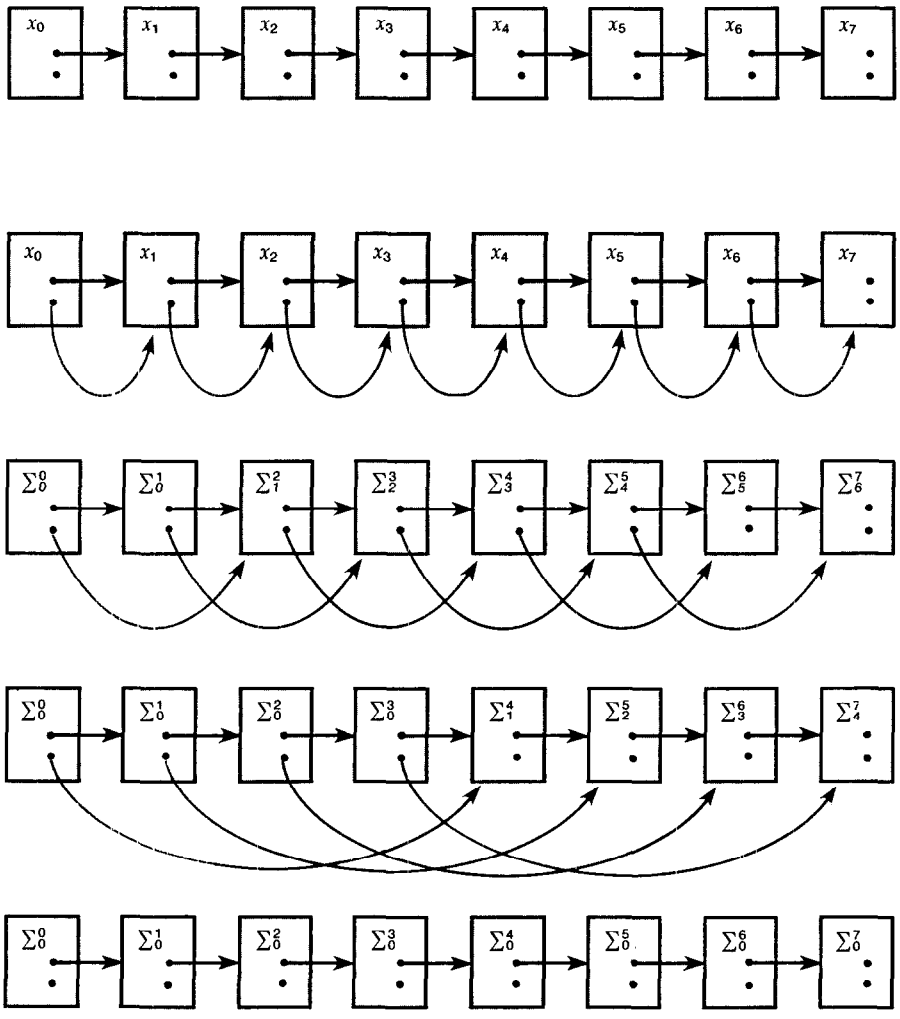


FIGURE 5. Computing Prefix Sums of a Serially Linked List

```
for all k in parallel do
  friend[k] := null
od
friend[list1] := list2
friend[list2] := list1
for all k in parallel
  chum[k] := next[k]
  while chum[k] ≠ null do
    if friend[k] ≠ null then
      friend[chum[k]] := chum[friend[k]]
    chum[k] := chum[chum[k]]
    fi
  od
od
```

The first part of the above algorithm is initialization: The component named *friend* is initialized to *null* in every cell; then the first cells of the two lists are introduced, so that they become *friends*. The second part plays the familiar logarithmic *chums* game, but at every iteration, a cell that has both a *chum* and a *friend* will cause its *friend's chum* to become its *chum's friend*. Believe it or not, when the dust has finally settled, the desired result does appear.

This algorithm has three additional interesting properties: First, it is possible to match up two lists of unequal length; the algorithm simply causes each extra cell at the end of the longer list to have no *friend* (that is, a *null friend*) (see Figure 6). Second, if, in the initialization, one makes the first cell of *list2* the *friend* of the first cell of *list1*, but not vice versa, then at the end all the cells of *list1* will have pointers to their *friends*, but the cells of *list2* are unaffected (their *friend* components remain *null*). Third, like the other linked-list algorithms, this one can process many lists (or pairs of lists) simultaneously.

With this primitive, one can efficiently perform such operations as componentwise addition of two vectors represented as linked lists.

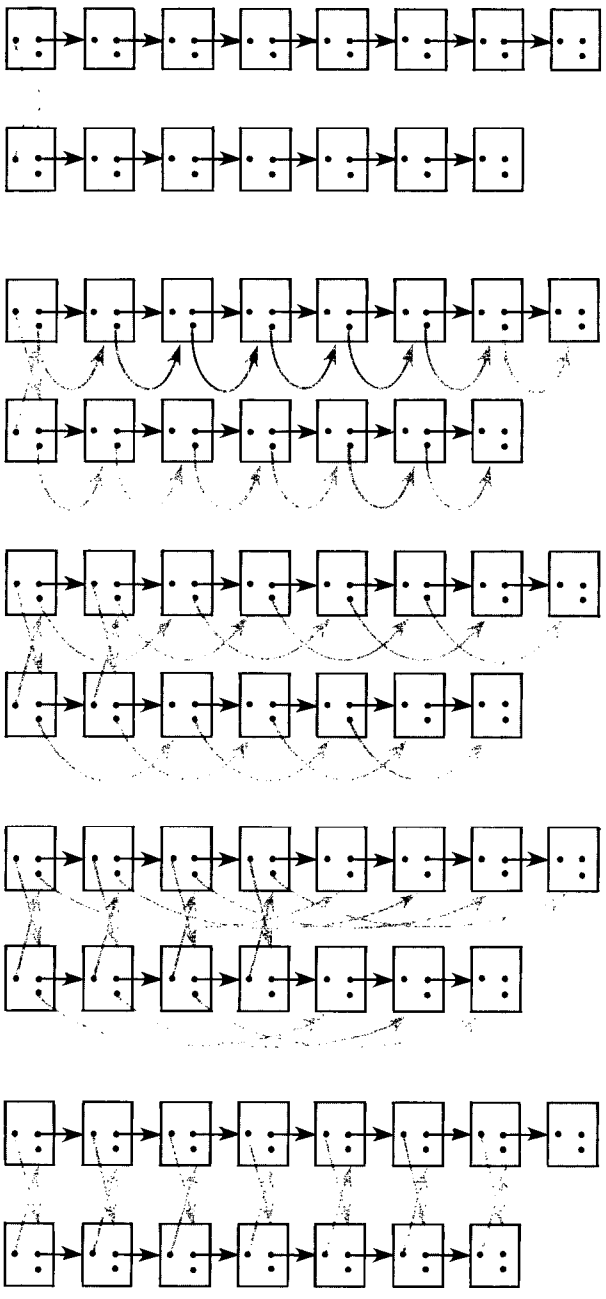


FIGURE 6. Matching Up Components of Two Lists

Region Labeling

How are linked-list operations used in practical applications? One such application is region labeling, where, given a two-dimensional image (a grid of pixel values), one must identify and uniquely label all regions. A *region* is a maximal set of pixels that are connected and all have the same value. Each region in the image must be assigned a different label, and this label should be distributed to every pixel in the region.

An obvious approach is to let each processor of a parallel computer hold one pixel. On a parallel computer with N processors communicating in a fixed two-dimensional pattern, so that each processor can communicate directly only with its four neighbors, this problem can be solved simply for an N-pixel image in the following manner: Since every processor has an address and knows its own address, a region will be labeled with the largest address of any processor in that region. To begin with, let each processor have a variable called *largest*, initialized to its own address, and then repeat the following step until there is no overall change of state. Each

processor trades *largest* values with all neighbors that have the same pixel value, and replaces its own *largest* value with the maximum of its previous value and any values received from neighbors. The address of the largest processor in a region therefore spreads out to fill the region.

Although the idea is simple, the algorithm takes time $O(\sqrt{N})$ in simple cases, and time $O(N)$ in the worst case (for images that contain a long "snake" that fills the picture). Lim [19] has devised algorithms for the Connection Machine that use linked-list techniques to solve the problem in time $O(\log N)$. In one of these algorithms, the basic idea is that every pixel can determine by communication with its two-dimensional neighbors whether it is on the boundary of its region, and, if so, which of its neighbors are also on the boundary. Each boundary pixel creates a pointer to each neighbor that is also a boundary pixel, and voilà: Every boundary has become a linked list (actually, a doubly linked list) of pixels. If the processor addresses are of the obvious form $x + Wy$, where x and y are two-dimensional coordinates and W is the width of the image, then the processor with the largest address for any region will be on its boundary. Using logarithmic-time linked-list algorithms, all the processors on a region boundary can agree on what the label for the region should be (by performing a maximum reduction on the linked list and then spreading the result back over the list). Since all the boundaries can be processed in parallel, it is then simply a matter of propagating the labels inward from boundaries to interior pixels. This is accomplished by a process similar to a parallel-prefix computation on the rows of the image. (There are many nasty details having to do with orienting the boundaries so that each boundary pixel knows which side is the interior and handling the possibility that regions may be nested within other regions, but these details can also be handled in logarithmic time.)

This application has the further property that the linked-list structure is not preexistent; rather, it is constructed dynamically as a function of the content of the image being processed. There is therefore no way to cleverly allocate or encode the structure ahead of time (e.g., as an array). The general communication facility of the Connection Machine model is therefore essential to the efficient execution of the algorithm.

Recursive Data Parallelism

We have often found situations where data parallelism can be applied recursively to achieve multiplicative effects. To multiply together a long chain of large matrices (a commonplace calculation in the study of systems modeled by Markov processes), we can use the associative scan operation to multiply together N matrices with $\log N$ matrix multiplications. In each matrix multiplication, the opportunity for parallelism is obvious, since matrix multiplication is defined in terms of operations on vectors. Another possibility would be to multiply the matrices using a systolic array-type algorithm [18], which will always run efficiently on a computer of the Connection Machine type. If the matrices are sparse, then we use the Pan–Reif algorithm [21], a data parallel algorithm that multiplies sparse matrices represented as trees. This algorithm fits well on a fine-grained parallel computer as long as it has capabilities for general communications. If the entries of the matrices contain high-precision numbers, there is yet another opportunity for parallelism within the arithmetic operations themselves. For example, using a scan-type algorithm for carry propagation, we can add two n-digit numbers in $O(\log n)$ time. Using a pipelined carry-save addition scheme [17], we can multiply in linear time, again by performing operations on all the data elements (digits) in parallel.

Summary and Conclusions

In discussing what kinds of computations are appropriate for data parallel algorithms, we initially assumed—when we began our work with the Connection Machine—that data parallel algorithms amounted to very regular calculations in simulation and search. Our current view of the applicability of data parallelism is somewhat broader. That is, we are beginning to suspect that this is an appropriate style wherever the amount of data to be operated upon is very large. Perhaps, in retrospect, this is a trivial observation in the sense that, if the number of lines of code is fixed and the amount of data is allowed to grow arbitrarily, then the ratio of code to data will necessarily approach zero. The parallelism to be gained by concurrently operating on multiple data elements will therefore be greater than the parallelism to be gained by concurrently executing lines of code.

One potentially productive line of research in this area is searching for counterexamples to this rule: that is, computations involving arbitrarily large data sets that can be more efficiently implemented in terms of control parallelism involving multiple streams of control. Several of the examples presented in this article first caught our attention as proposed counterexamples.

It is important to recognize that this question of

programming style is not synonymous with the hardware design issue of MIMD versus SIMD computers. MIMD computers can be well suited for executing data parallel programs: In fact, depending on engineering details like the cost of synchronization versus the cost of duplication, they may be the best means of executing data parallel programs. Similarly, SIMD computers with general communication can execute control-style parallelism by interpretation. Whether such interpretation is practical depends on the details of costs and requirements. While interesting and important in their own right, these questions are largely independent of the data parallel versus control parallel programming styles.

Having one processor per data element changes the way one thinks. We found that our serial intuitions did not always serve us well in parallel contexts. For example, when sorting is fast enough, the order in which things are stored is often unimportant. Then again, if searching is fast, then sorting may be unimportant. In a more general sense, it seems that the selection of one data representation over another is less critical on a highly parallel machine than on a conventional machine since converting all the memory from one representation to another does not take a large amount of time. One case where our serial intuitions misled us was our expectation that parallel machines would dictate the use of binary trees [14]. It turns out that linear linked lists serve almost as well, since they can be easily converted to balanced binary trees whenever necessary and are far more convenient for other purposes.

Our own transition from serial to parallel thinkers is far from complete, and we would be by no means surprised if some of the algorithms described in this article begin to look quite "old-fashioned" in the years to come.

REFERENCES

1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs (1977 ACM Turing Award Lecture). *Commun. ACM 21*, 8 (Aug. 1978), 613–641.
2. Batcher, K.E. Sorting networks and their applications. In *Proceedings of the 1968 Spring Joint Computer Conference* (Reston, Va., Apr.) AFIPS, Reston, Va., 1968, pp. 307–314.
3. Batcher, K.E. Design of a massively parallel processor. *IEEE Trans. Comput. C-29*, 9 (Sept. 1980), 836–840.
4. Bawden, A. A programming language for massively parallel computers. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., Sept. 1984.
5. Bawden, A. Connection graphs. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*. ACM, (Cambridge, Mass., Aug. 4–6). New York, 1986, pp. 258–265.
6. Blelloch, G. AFL-I: A programming language for massively concurrent computers. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., June 1986.
7. Blelloch, G. Parallel prefix versus concurrent memory access. Tech. Rep., Thinking Machines Corp., Cambridge, Mass., 1986.
8. Bouknight, W.J., Denenberg, S.A., McIntyre, D.E., Randall, J.M., Sameh, A.H., and Slotnick, D.L. The ILLIAC IV system. *Proc. IEEE 60*, 4 (Apr. 1972), 369–388.
9. Christman, D.P. Programming the Connection Machine. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., Jan. 1983.
10. Christman, D.P. Programming the Connection Machine. Tech. Rep. ISL-84-3, Xerox Palo Alto Research Center, Palo Alto, Calif., Apr. 1984. (Reprint of the author's master's thesis at MIT.)
11. Falkoff, A.D., and Orth, D.L. Development of an APL standard. In *APL 79 Conference Proceedings* (Rochester, N.Y., June). ACM, New York, pp. 409–453. Published as *APL Quote Quad 9*, 4 (June 1979).
12. Flanders, P.M., et al. Efficient high speed computing with the distributed array processor. In *High Speed Computer and Algorithm Organization*, Kuch, Lawrie, and Sameh, Eds. Academic Press, New York, 1977, pp. 113–127.
13. Haynes, L.S., Lau, R.L., Siewiorek, D.P., and Mizell, D.W. A survey of highly parallel computing. *Computer* (Jan. 1982), 9–24.
14. Hillis, W.D. *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.
15. Iverson, K.E. *A Programming Language*. Wiley, New York, 1962.
16. Knuth, D.E. *The Art of Computer Programming*. Vol. 3, *Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
17. Knuth, D. E. *The Art of Computer Programming*. Vol. 2, *Seminumerical Algorithms (Second Edition)*. Addison-Wesley, Reading, Mass., 1981.
18. Kung, H.T., and Lieserson, C.E. Algorithms for VLSI processor arrays. In *Introduction to VLSI Systems*, L. Carver and L. Conway, Eds. Addison-Wesley, New York, 1980, pp. 271–292.
19. Lim, W. Fast algorithms for labeling connected components in 2-D arrays. Tech. Rep. 86.22, Thinking Machines Corp., Cambridge, Mass., July 1986.
20. Minsky, M., and Papert, S. *Perceptrons*. 2nd ed. MIT Press, Cambridge, Mass., 1979.
21. Pan, V., and Reif, J. Efficient parallel solution of linear systems. Tech. Rep. TR-02-85, Aiken Computation Laboratory, Harvard Univ., Cambridge, Mass., 1985.
22. Schwartz, J.T. Ultracomputers. *ACM Trans. Program. Lang. Syst. 2*, 4 (Oct. 1980), 484–521.
23. Shaw, D.E. *The NON-VON Supercomputer*. Tech. Rep., Dept. of Computer Science, Columbia Univ., New York, Aug. 1982.
24. Steele, G.L., Jr., and Hillis, W.D. Connection machine Lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (Cambridge, Mass., Aug. 4–6). ACM, New York, 1986, pp. 279–297.
25. Turner, D.A. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9* (1979), 31–49.

CR Categories and Subject Descriptors: B.2.1 [**Arithmetic and Logic Structures**]: Design Styles—*parallel*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; E.2 [**Data Storage Representations**]: *linked representations*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; G.1.0 [**Numerical Analysis**]: General—*parallel algorithms*

General Terms: Algorithms

Additional Key Words and Phrases: Combinator reduction, combinators, Connection Machine computer system, log-linked lists, parallel prefix, SIMD, sorting, Ultracomputer

Authors' Present Address: W. Daniel Hillis and Guy L. Steele, Jr., Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214.