Ultracomputers

J. T. Schwartz
New York University

A class of parallel processors potentially involving thousands of individual processing elements is described. The architecture is based on the perfect shuffle connection and has two favorable characteristics: (1) Each processor communicates with a fixed number of other processors. (2) Important communication functions can be accomplished in time proportional to the logarithm of the number of processors. A number of basic algorithms for these "ultracomputers" are presented, and physical design considerations are discussed in a preliminary fashion.

Key Words and Phrases: parallelism, parallel computation, parallel algorithms
CR Categories: 5.25, 6.21

1. THE OPPORTUNITY

Integrated circuit technology is evolving continuously and rapidly toward smaller elementary devices and denser, more complex functions on each single silicon chip. New processing and lithographic techniques may make it possible to fabricate chips containing 10^7 and 10^8 individual transistors. One such chip would contain more function than today's largest computers. The problem of utilizing this potentially enormous computing power effectively is steadily becoming more significant.

However, *the mere possibilities inherent in very-large-scale integration (VLSI) do not themselves create adequate ways of exploiting this technology.* The programming techniques now current, which are still shaped by the vanishing era in which computing power was a scarce resource to be used sparingly, are ill-suited to the opportunities opening up. Few significant architectural concepts reflecting the LSI potential have as yet appeared. Rapidly improving technology has generally been used to map established computer architectures into smaller and cheaper formats. Thus the milestones of progress have been 16K and 64K memory chips; 8-bit fast adders, shifters, and multipliers; 16-bit microcomputer chips; etc. This line of development will clearly culminate in a powerful 32-bit computer realized on just a few chips, but few new ways of using computational power have emerged from this development.

How then can we hope to exploit LSI technology more adequately?

This work was supported in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under Contract DE-AC02-76ER03077 and in part by the National Science Foundation under Grant NSF-MCS76-00116.

Author's address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012.

A central problem is to *develop techniques which allow organization of concurrent computation on a very large scale.* Until recently, the cost of high-performance computing devices has been large enough to make it impractical to use more than a few such devices in a single computing system. Microstructure technology now begins to make it possible to put together *highly concurrent computing assemblages* consisting of thousands of interconnected elements, each as powerful as one of today's large computers. The deepest opportunities inherent in microstructure technology can only be realized if we find effective ways of structuring and using such assemblages.

Any attempt to do this will raise the following questions:

(1) *Given an indefinitely large supply of small but powerful computers, what is the best way to connect them?* Physical limitations are likely to imply that each computer in such an assemblage can only be connected to a limited number of others. Then, given a problem to be solved using a concurrent assemblage, how can an effective interconnection pattern be found? Are any interconnection patterns generally appropriate for solving extensive classes of problems?

(2) *What fundamental computational operations and what "concurrent algorithms" can exploit large-scale parallelism?* In learning to use serial computers effectively, we have come to recognize certain computational operations (e.g., sorting, hashing and hashed search, tree manipulation, the fast Fourier transform and its variants) as fundamental and have discovered surprisingly efficient algorithms for implementing these operations. Our ability to make effective use of large-scale parallelism will depend on corresponding parallel algorithms.

(3) *What languages are appropriate for large-scale parallel programming?* Given an algorithm, we can proceed to realize it on a serial computer, in assembly language if necessary. But even for a serial processor, this approach is inadequate, error prone, and productive of inextensible, hard-to-design, hard-to-read programs rigidly adapted to some single application. Programming only reaches maturity when general algorithms and structures adequate for broad classes of applications are absorbed into the logical structure of powerful, general programming languages. The *semantic framework* defined by such languages focuses our thinking about applications and greatly facilitates application development. In writing programs for highly parallel devices, an undisciplined assembly-language programming approach is bound to be even more nightmarish than in the serial case. Therefore we must define (necessarily general) higher level languages in order to be able to control large parallel assemblages. A significant goal here is to find languages whose dictions exploit the concurrency of such assemblages *implicitly*, thus freeing the programmer from *explicit* concern with the details of interprocessor coordination.

(4) *What are the theoretical limits of effectiveness for parallel assemblages?* In the serial case, the design of algorithms has come to be illuminated by a growing body of theoretical knowledge concerning the ultimate limits of algorithm performance. Thus, for example, we know that an array of N elements cannot be sorted in fewer than N serial steps (since each element of the array must be examined at least once); a somewhat more refined analysis shows that for certain general classes of comparison sorts, at least $N \log N$ steps are necessary. We know also that certain classes of difficult (NP-complete) problems can probably not be solved in polynomial time, and that certain other still more totally intractable

problems cannot be solved in less than hyperexponential time. These results, fragmentary as they are, have begun to provide fundamental yardsticks against which the quality of specific serial algorithms is measured. Until a like body of theoretical knowledge has been developed for highly concurrent algorithms, we will have little basis for judging the extent to which a given concurrent approach can be improved.

2. THEORETICAL LIMITATIONS ON PARALLEL COMPUTATION

A scattering of results on the theoretical limits of parallel computation is found in the literature, but what little is known is still fragmentary and inadequate. Nevertheless, to orient ourselves it is worth making a few easy observations on the limits of effectiveness of highly concurrent computing assemblages. In the serial case, an obvious but useful remark is that no algorithm which reads J items of input and produces K items of output can operate in less than $N = J + K$ cycles. Algorithms which attain this maximum possible level of efficiency are often called *linear*; other "good" algorithms, which often have a recursive internal structure, come close to this limit, for example, can process N items in $N \log N$ cycles. Usable but less adequate algorithms, which must iterate more often over an input collection of items, may require N^2, N^3, $N^{2.81}$, etc., cycles to complete.

We can expect similar distinctions to arise in the concurrent case, but here it seems that we must distinguish a richer range of possibilities. At the most favorable extreme, certain problems will be totally decomposable into completely independent operations which can proceed in parallel. If enough processors are available to devote one processor to each such operation, then all operations can be accomplished in a number of cycles independent of the number N of operations to be performed. The hardware prototypes of such algorithms, which can be called *constant time* concurrent algorithms, are the familiar bit-parallel Boolean operations.

At the opposite extreme, there exist computational processes which are *completely unparallelizable*, that is, can be performed no faster by an indefinitely large assemblage of processors than by a single processor. To see this, we need a more precise model of parallel computation. One such model is that in which a very large number N of identical processors (each with a conventional order-code set) share a common memory which they can read simultaneously in a single cycle. In such a model we also assume that during any access cycle any number of processors can simultaneously write to memory, and that a memory cell to which writes are simultaneously addressed by many processors will come to contain some one of the quantities written into it (perhaps a randomly selected one of these quantities, or perhaps their minimum). We call this very general model of parallel computation the *paracomputer* model. Though structures of this sort are in fact not realizable because of physical fan-in limitations, they can play a useful role as theoretical yardsticks for measuring the limits of parallel computation. In this connection it is useful to note that a single processor with a large enough memory can clearly simulate an N-processor paracomputer in time $O(N)$; thus an N-processor paracomputer can never be more than N times as fast as a serial computer. Moreover, there do exist computations which cannot be performed more quickly on an N-processor paracomputer than on a serial

processor. A simple example of this kind, described by Kung [16], is the task of raising a number x to a large power x^{2^k}. On a serial computer this can be done in k cycles by successive squarings. It cannot be done any faster on a paracomputer, since no matter how many processors are available, no power of x higher than x^2 can be formed in one cycle of operation, none higher than x^4 in two cycles, etc. This example shows clearly that a central problem of parallel computation is to understand the range of processes for which there exist parallel algorithms which attain something close to the efficiency of optimal serial algorithms.

Ultracomputers. Though theoretically useful, the paracomputer model is unrealizably powerful. As already remarked, in a physically realizable assemblage we cannot expect any computing element to have more than some fixed number of external connections. We must therefore consider parallel assemblages in which a large number N of communicating processors, each with its own memory, are connected together, but where each processor communicates with a fixed number k of other processors. To these significantly more limited parallel assemblages, and in particular to those which use the favorable interconnection pattern described in the next section, we give the name *ultracomputer.*

It is clear that an N-processor ultracomputer cannot compute more rapidly than an N-processor paracomputer. Whether it can do as well in handling any particular problem depends on the communication pattern characteristic of the problem. Note in particular that many problems of size N will have N outputs, at least one of which depends on all N inputs.

Since in an ultracomputer we cannot bring more than k quantities together in one place during any cycle of assemblage operation, the solution of such a problem will clearly require at least $\log_k N$ cycles of operation. This is an attainable limit: In what follows, we will review many interesting $O(\log N)$ ultracomputer algorithms. (The hardware prototype of these algorithms, which can be called *logarithmic* concurrent algorithms, is carry-jumping addition of N-bit quantities.)

The performance to be expected from an ultracomputer will depend not only on the problem to which it is applied but on the relationship between the mass of data to be processed and the number of processors available. No matter how large the computational assemblage we envision, we will generally have to deal with problems or subproblems consisting of $M > N$ objects when only N processors are available. In favorable cases, by processing the objects in batches we can hope to complete all necessary steps in $\pi \cdot (M/N)$ steps, where π is the time required to process N objects of an N-processor concurrent assemblage.

Next consider the case in which we have fewer data objects than available processors. Here it is appropriate to remark that the relative inefficiency of computational procedures which apply to structures S of size N but require time nonlinear in N can often be ascribed to the fact that such procedures must interrelate the component parts of these structures and hence must examine a substantial part of the Cartesian product $S \times S$ (as in sorting or matrix multiplication), or $S \times S \times S$, or even larger objects such as the power set $\text{pow}(S)$ of S. If in such cases the number of processors available in a concurrent assemblage exceeds N, then we can hope to use these additional processors to accelerate the computation to be applied to S. In what follows we will review several cases of this sort. Often the sensitivity of the computation rate to the number of data items to be processed is quite significant. For example, as shown recently by

Nassimi and Sahni [18], $N^{1-\epsilon}$ elements can be sorted in time $O(\epsilon^{-1}\log N)$ using an N-processor ultracomputer, whereas arrays of size N seem to require time $O(\log^2 N)$.

For highest efficiency it is often important to balance processor intercommunication with local processing. Consider an algorithm, such as that for computing the sum or product of a set of numbers, which would process N data items in time $O(N)$ using a single processor. As already noted, the time required to develop the same result in an N-processor ultracomputer cannot be less than $O(\log N)$. The appearance of the factor $\log N$ seems to indicate a (mild) inefficiency, since $O(\log N)$ processors seem to be required to do the work of one. However, in many such cases we are able to regard the "inefficiency factor" $\log N$ as resulting from an imbalance between the amount of work performed locally within each processor and the amount of interprocessor communication performed. Then, if the result to be developed is significantly smaller than the mass of input data being processed, we may be able to restore balance by giving each processor a larger amount of local work and only communicating "summarized" or "preprocessed" data. For example, N integers spread among the N processors of an ultracomputer can be totaled in time $O(\log N)$ by a well-known process described just below. Thus, if we spread $N \log N$ integers among the same N processors, $\log N$ integers to each processor, we can proceed as follows:

(a) in $\log N$ steps, develop a local total in each processor;
(b) in $O(\log N)$ additional steps, develop a grand total.

Overall, we will then have totaled $N \log N$ integers in time $O(\log N)$ using N processors, attaining an efficiency of processor use which is directly proportional to single-processor efficiency.

An algorithm of this sort, which uses N processors in an asymptotically efficient way, can be said to be *completely parallelizable.* In some cases we may wish to emphasize the amount of local work W which must be assigned to each processor to achieve complete parallelizability; in such cases we may speak of an algorithm as being *completely parallelizable at level W.* In this sense we can speak of addition of a collection of numbers as being "completely parallelizable at level $\log N$." These ideas are studied by Gottlieb and Kruskal [12].

3. THE PERFECT SHUFFLE INTERCONNECTION

We saw in the preceding section that no output which depends on all of N inputs can be realized in an N-processor ultracomputer in fewer than $O(\log N)$ steps. Interconnection patterns which allow large classes of such outputs to be developed in $O(\log N)$ steps are therefore of special interest. In view of the importance of data communication operations, patterns which allow all permutations of data among processors to be realized in $O(\log N)$ processing cycles are particularly to be sought. These significant goals are all achieved by the *perfect shuffle* interconnection pattern which goes back to Clos [6] and Benes [4], but which was further explored and perfected by Stone [27]; see also Pease [22, 23].

This interconnected processing assemblage involves $N = 2^D$ processors p_n. Each processor is assumed to be able to access both its own memory and the memory of four other processors to which it is connected, namely, p_{n+1}, p_{n-1} (where sums

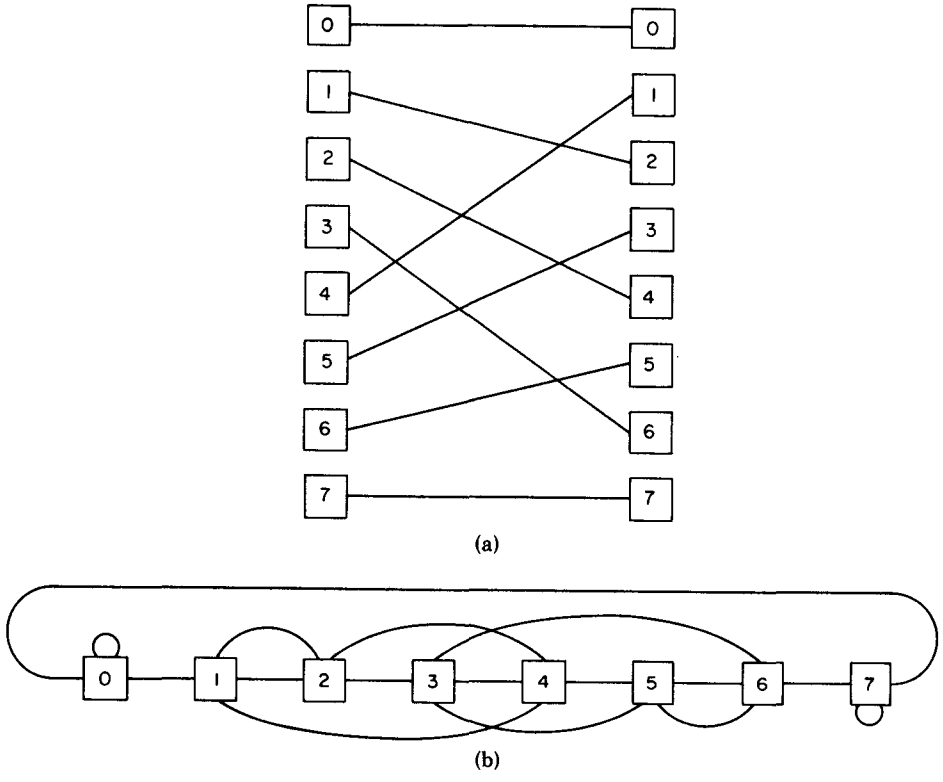

Fig. 1. (a) Logical schematic of perfect shuffle connections among eight processors. (b) Connections in an eight-processor ultracomputer.

are taken modulo N), $p_{\sigma(n)}$, and $p_{\sigma^{-1}(n)}$. Here the fundamental *shuffle map* $\sigma(n)$ is defined by

$$\sigma(n) = \textbf{if } n < N/2 \textbf{ then } 2n \textbf{ else } 2n - N + 1, \tag{1}$$

so that the inverse *unshuffle map* $\sigma^{-1}(n)$ is defined by

$$\sigma^{-1}(n) = \textbf{if } n \text{ is even } \textbf{then } n/2 \textbf{ else } (n-1)/2 + N/2. \tag{1'}$$

If we let $\beta_D \cdots \beta_1$ denote the binary representation of n, then

$$\sigma(\beta_D \cdots \beta_1) = \beta_{D-1}\beta_{D-2} \cdots \beta_1\beta_D, \tag{2}$$

$$\sigma^{-1}(\beta_D \cdots \beta_1) = \beta_1\beta_D\beta_{D-1} \cdots \beta_2. \tag{3}$$

We can describe this interconnection pattern, which is shown in Figure 1, in the following amusing way (which also explains its name). Suppose that the processor p_n corresponds to the nth card in an N-card deck, and that we divide the deck exactly in half and then shuffle perfectly, putting each card from the low (numbered) half of the deck into an even position, and each card in the high (numbered) half of the deck into an odd position. This gives exactly the shuffle transformation σ.

An important property of the shuffle interconnection can be seen as follows. Many algorithms having a recursive, "divide and conquer" character can be

Initial data	Phase 1	Phase 2	Phases 3–6	Phase 7	Phase 8	Phase 9
	Add	Unshuffle		Add	Shuffle	Add
X_0	X_0	X_0	Phases	X_0	X_0	X_0
X_1	$X_0 \otimes X_1$	X_2	3–6 omitted	X_2	$X_0 \otimes X_1$	$X_0 \otimes X_1$
X_2	X_2	X_4	here	X_4	X_2	$X_0 \otimes X_1 \otimes X_2$
X_3	$X_2 \otimes X_3$	X_6		X_6	$X_0 \otimes \cdots \otimes X_3$	$X_0 \otimes \cdots \otimes X_3$
X_4	X_4	$X_0 \otimes X_1$		$X_0 \otimes X_1$	X_4	$X_0 \otimes \cdots \otimes X_4$
X_5	$X_4 \otimes X_5$	$X_2 \otimes X_3$		$X_0 \otimes \cdots \otimes X_3$	$X_0 \otimes \cdots \otimes X_5$	$X_0 \otimes \cdots \otimes X_5$
X_6	X_6	$X_4 \otimes X_5$		$X_0 \otimes \cdots \otimes X_5$	X_6	$X_0 \otimes \cdots \otimes X_6$
X_7	$X_6 \otimes X_7$	$X_6 \otimes X_7$		$X_0 \otimes \cdots \otimes X_7$	$X_0 \otimes \cdots \otimes X_7$	$X_0 \otimes \cdots \otimes X_7$

Fig. 2. Ultracomputer summation procedure, $N = 8$.

implemented using a first step *involving nearest neighbor communication only*, which decomposes the initial problem into two independent parts, one of which involves only data in even processors, the other only data in odd processors. When this is possible, the (reversed) perfect shuffle interconnections can be used to move data, in parallel, from all the even processors to the low-numbered processors, and from all the odd processors to the high-numbered processors. This allows nearest neighbor interactions to be used again during the next cycle of processing and hence allows parallel processing to continue. It is then clear that, at least in favorable cases, an overall result can be produced in time $O(D) = O(\log N)$.

This idea can be put in another way which can serve to stress its importance. The shuffle moves data from $p_n = p_{\beta_D \cdots \beta_1}$ to $p_{\sigma(n)} = p_{\beta_{D-1}\beta_{D-2}\cdots\beta_1\beta_D}$. Thus, if we shuffle data j times and then let nearest neighbors communicate, the data originally present in p_n are brought into contact with data originally present in p_{n+2^j}. This makes it plain that the shuffle connection is just as effective as the more expensive set of "hypercube" connections (which would connect p_n to all of $p_{n+2^0}, p_{n+2^1}, p_{n+2^2}, \ldots$), at least for algorithms which use these connections either in strictly ascending or strictly descending order.

Summing, Summing by Groups, Taking. As an example of the foregoing, suppose that an item w_n of data is present in each processor p_n of our ultracomputer, and moreover that an associative operation $a \otimes b$ in terms of which these data items can be combined is available. The elementary operation of *summing* then calculates the sequence $w_0, w_0 \otimes w_1, \ldots, w_0 \otimes \cdots \otimes w_{N-1}$ and deposits the nth of these quantities in p_n. (Note that we count both processors and data items starting with 0 rather than 1.) This fundamental operation can be realized in time $O(\log N)$. To do this, we can proceed as follows:

(a) Replace w_n by $w_{n-1} \otimes w_n$ for each odd n.

(b) Proceeding recursively, apply summing to the odd elements. (This can be accomplished by first unshuffling, then applying summing to the upper half of the processors, then shuffling.) At the end of this step, every odd processor p_n will contain $w_0 \otimes \cdots \otimes w_n$.

(c) Replace w_n by $w_{n-1} \otimes w_n$ for every even $n > 0$.

Figure 2 illustrates the action of this easy but important parallel summation procedure in an eight-processor ultracomputer.

Table I. Ultracomputer Algorithms

Process	Time	Reference
(1) Arbitrary permutation of N elements (given appropriate "permutation descriptor")	$O(\log N)$	Clos [6], Benes [4]
(2) All partial sums, etc., of N elements	$O(\log N)$	Pease [23], Stone [27]
(3) Separating and packing marked data	$O(\log N)$	
(4) Merging two lists of N items	$O(\log N)$	Batcher [2]
(5) Sorting N items	$O(\log^2 N)$	Batcher [2]
(6) Sorting $N^{1-\epsilon}$ items	$O(\log N)$	Nassimi and Sahni [18]
(7) Set union, intersection, difference, application of single-valued maps	$O(\log^2 N)$	
(8) Interprocessor message transmission in general patterns	$O(\log^2 N)$	
(9) Fast Fourier transform	$O(\log N)$	Pease [23], Stone [27]
(10) Multiplication of matrices of size $(N \log N \log\log N)^{1/2.81}$	$O(\log N \log\log N)$	Chandra [5]
(11) Inversion of matrices of size $N^{1/2.81}$	$O(\log^2 N)$	Csanky [7]
(12) Calculation of permutation descriptor	$O(\log^4 N)$	Opferman and Tsao-Wu [20]
(13) Connected components of an undirected graph	$O(\log^4 N)$	Levitt and Kautz [17], Hirschberg [13]

Several other very useful operations are closely related to the summing operation that we have just described and can be realized by summing. Suppose, in particular, that certain of the data items w_n are specially flagged. We can regard the flagged items as dividing the set of all processors into *groups* $p_\mu, \ldots, p_\nu$, each of which consists of a processor containing a flagged item and all following processors up to but not including that containing the next flagged item. Then the operation of *summing by groups* forms the quantities $w_\mu \otimes w_{\mu+1} \otimes \cdots \otimes w_\nu$ and deposits these quantities within the successive processors of a group. To accomplish this operation, we have only to put $a \otimes' b = a \otimes b$ if b is unmarked; otherwise $a \otimes' b = b$. Moreover, we let $a \otimes' b$ be marked if a or b is marked. Then using the operation $\otimes'$ and summing yields the same result as if we sum by groups using the operation $\otimes$.

The operation of *taking* puts the first flagged item to the left of w_n into w_n. This is simply a matter of using the operator $a \otimes b = a$ and summing by groups.

The taking and summing operations can obviously be used to place, into each of the processors of the group $p_\mu, \ldots, p_\nu$, $\min(w_\mu, \ldots, w_\nu)$, $\max(w_\mu, \ldots, w_\nu)$, $w_\mu + \cdots + w_\nu$, the exclusive-Or of $w_\mu, \ldots, w_\nu$ (regarded as bit strings), etc.

Table I lists many computationally important processes which can be performed at high speed on a (shuffle-connected) ultracomputer. Several of the algorithms listed in Table I are reviewed in this paper; for others we refer the reader to the sources enumerated in the table and in the appended bibliography.

4. A PRELIMINARY ULTRACOMPUTER PROGRAMMING LANGUAGE

To use ultracomputers effectively, high-level programming languages will certainly be required. The design of such languages raises difficult problems, which we do not intend to pursue in this paper. Nevertheless, in order to be able to

represent our parallel algorithms as programs, it is useful to define at least a highly simplified ultracomputer language, capable of describing that important subclass of ultracomputer procedures (including most of those to be considered in this paper) in which all the processors of an ultracomputer act in approximately synchronous fashion. To do this, we make informal use of a Fortran- or PASCAL-like language whose variables are all vectors of as many components as the ultracomputer contains processors. We regard such vectors as being homogeneous and consider their components to be integers, reals, structured records, or arrays of integers or reals. In actual programs we may write the number $N = 2^D$ of processors in the array as $N = 2{**}D$. The reserved constant name ID will designate the vector whose components are $0, \ldots, N - 1$, the value n being assumed in the nth processor. A constant (e.g., 51) designates a vector of size N all of whose components have the indicated constant value.

In the following codes we generally assume that the processors of the ultracomputer are operating in a roughly synchronous mode, that is, all executing nearby instructions during each instruction cycle. Thus, for example, in writing the instruction sequence

```
DO J = 1 TO 100;
    A(J) = B(J) + C(J) * D(J) + X;
END DO;
```

we assume that

(a) J designates a vector of N integers, all of which are simultaneously initialized, advanced, tested, etc., during execution of the loop shown above. Likewise, X designates a vector of N reals.

(b) A, B, and C designate arrays of vectors of some declared extent DIM $\times N$ (e.g., $100 \times N$). All the separate additions and multiplications appearing in the above loop (e.g., the N-multiplications $C(J) * D(J)$ for a fixed value of J) are performed synchronously; likewise the N stores indicated by $A(J) = \cdots$ (for a fixed value of J).

The important point here is that *once synchronous operation is established, it will persist until disrupted by a conditional transfer involving a (vector-valued) expression whose components may differ.*

Next consider an IF statement, such as

```
IF ID ≥ N/2 THEN
    J = J + 1;
ELSE
    J = J − 1;
END IF;
```

The test with which this statement begins will affect certain of the processors one way and the remainder another, thus destroying synchronization. One will often wish to reestablish synchronized operation at the end of such a conditional block (and sometimes at other program points as well). For this purpose the special operation

SYNC

is provided. This operation delays the processors which execute it until all processors have arrived at the SYNC operation, at which time they all proceed in resynchronized fashion. (In the simple codes we consider, this could be done by precalculating the length of each path through the code and inserting an appropriate number of NO-OPS along the shorter paths.) Thus we can reestablish synchronization after the IF-statement shown above by placing a SYNC immediately after the END IF. Because this construction occurs commonly, we allow the combination

```
END IF;
SYNC;
```

to be abbreviated as

```
ENDS IF;
```

We use

LEFT I and RIGHT I

to denote the left-shift (respectively, right-shift) of the N-dimensional vector value of I among processors. Similarly, we use

SHUF I and ISHUF I

to designate the shuffled and inverse-shuffled value of I. The component of I held in the nth processor will be written as $I[n]$; thus

$$\begin{aligned}(\text{SHUF } I)[n] &= I[\sigma^{-1}(n)] \\ &= I[\textbf{if } \text{even}(n) \textbf{ then } n/2 \\ &\qquad \textbf{else } (n-1)/2 + N/2] \\ (\text{ISHUF } I)[n] &= I[\sigma(n)] \\ &= I[\textbf{if } n < N/2 \textbf{ then } 2n \\ &\qquad \textbf{else } 2n - N + 1]\end{aligned}$$

With these conventions we can write the summing algorithm described in the preceding section as follows:

```
PROC SUM(W, LB);      /* generate partial sums of w */
DECLARE W: REAL;
DECLARE LB: FIXED;
IF ODD(ID) & (LB < ID)
    W = LEFT W + W;
ENDS IF;
IF LB + 1 < N - 1 THEN
    W = ISHUF W;
    CALL SUM(W, (LB + N)/2);
    W = SHUF W;
ENDS IF;
IF EVEN(ID) & (LB < ID) THEN
    W = LEFT W + W;
END IF;
END PROC SUM;
```

We can readily remove recursion from (an optimized version of) the above algorithm and obtain the following code, in which

$$\text{LOWBITS(ID, } J) = \beta_J \cdots \beta_1,$$

where $\beta_D \cdots \beta_1$ denotes the binary representation of ID.

```
PROC SUM(W);
DECLARE W: REAL;
DECLARE J: FIXED;
DO J = D TO 1 BY -1;
    IF LOWBITS(ID, J) ≠ 0 THEN
        W = LEFT W + W;
    ENDS IF;
    W = ISHUF W
END DO;
END PROC SUM;
```

(Simulators for languages quite similar to the above have been implemented as PL/I extensions by Gottlieb [8, 9] and Gottlieb and Kruskal [10, 11] and have been used to check many of the algorithms reported in this paper.)

Fast Fourier Transform. Using the same style, a variant of the $O(\log N)$ parallel fast Fourier transform algorithm of Pease [22] and Stone [27] can easily be written. For the reader's convenience we summarize the algorithm before presenting the actual code. Given an array $c(n)$ of 2^D complex numbers, the FFT algorithm forms the discrete Fourier transform

$$b(m) = \sum_n \omega^{mn} c(n), \tag{4}$$

where ω is a primitive 2^Dth root of unity. The algorithm has the following structure. Let the binary representation of the integer m be $\beta_D \cdots \beta_1$, and define the permutation ρ of integers by $\rho(\beta_D \cdots \beta_1) = \beta_1 \cdots \beta_D$. Also, let set_j and $drop_j$ be defined by

$$set_j(\beta_D \cdots \beta_1) = \beta_D \cdots \beta_{j+1} 1 \beta_{j-1} \cdots \beta_1, \tag{5a}$$

$$drop_j(\beta_D \cdots \beta_1) = \beta_D \cdots \beta_{j+1} 0 \beta_{j-1} \cdots 1, \tag{5b}$$

and let

$$low_j(\beta_D \cdots \beta_1) = \beta_j \cdots \beta_1. \tag{6a}$$

Let $A^{(0)}(m) = c(m)$, and inductively let

$$A^{(j)}(m) = A^{(j-1)}(drop_j(m)) + \omega^{2^{D-j} low_j(m)} A^{(j-1)}(set_j(m)). \tag{6b}$$

Then it can be shown that $A^{(D)}(m) = b(m)$.

After $j-1$ successive unshuffles, m, $drop_j(m)$, and $set_j(m)$ (only two of which are different) will always lie in adjacent processors. Hence, if the D coefficients $\omega^{2^{D-j} low_j(\sigma^{j-1}(m))}$ are precalculated and available in processor p_m, each of the D successive operations (6a and b) can be performed in completely parallel fashion.

Clearly $A^{(0)}$ can be formed from the array c by a data permutation. Thus the

Fourier transform of an N-element array can be formed in a number of cycles proportional to log N and is therefore completely parallelizable.

Next we give detailed code for the fast Fourier transform.

```
PROC FFT(Z);      /*discrete Fourier transform*/
DECLARE Z, FFT: COMPLEX;
DECLARE Y, OMEGA: COMPLEX;
DECLARE J, K, ID_SHUF, POW2: FIXED;
DECLARE COEF: (D)COMPLEX;      /* precalculated coefficients */
                               /* mentioned above */
/* the following initialization code need only be executed once */
OMEGA = EXP(2πi/N);
ID_SHUF = ID;
DO J = 1 TO D;
    COEF(J) = OMEGA**(LOWBITS(ID_SHUF, J)*2**(D - J));      /* apply σ */
    IF ID_SHUF > (N - 1)/2 THEN
       ID_SHUF = 2*ID_SHUF - N + 1;
    ELSE
       ID_SHUF = 2*ID_SHUF
    END IF;
END DO;     /* end of initialization code */
/* now we calculate y[m] = x[ρ(m)]. We will show later that any data permutation can be
   realized in 4D - 3 cycles. The loop shown below constitutes a slightly more efficient
   realization of the permutation ρ; 3.5D + 1 cycles are used */
Y = Z;
DO K = 1 TO 3;
    IF ODD(D) AND K = 2 THEN
        POW2 = N/4;
    ELSE
        POW2 = N/2;
    END IF
    DO J = 1 TO D/2;
        IF EVEN(ID/POW2) THEN
            Y = SHUF Y;
        ELSE IF EVEN(ID) THEN
            Y = SHUF RIGHT Y;
        ELSE
            Y = SHUF LEFT Y;
        ENDS IF
    END DO;
    IF ODD(D) AND K ≠ 2 THEN
        Y = ISHUF Y
    ENDS IF;
END DO;
DO K = 1 TO D/2;
    Y = ISHUF Y;
END DO;
/* after the preceding data permutation has been carried out, the arithmetic part of the
   Fourier transform is simple: */
DO K = 1 TO D;
    IF EVEN(ID) THEN
        Y = Y + COEF(K)*RIGHT Y;
```

```
    ELSE
        Y = LEFT(Y) + COEF(K) * Y;
    ENDS IF
    Y = ISHUF Y;
END DO;
RETURN Y;
END FFT;
```

5. DATA REARRANGEMENT AND COMMUNICATION ALGORITHMS

5.1 Packing and Segregating Data Items

Suppose that the elements of some ordered set s are dispersed among the processors p_n of an ultracomputer, with at most one element per processor. We suppose that these elements occur in the order in which we wish to keep them, but that elements of s are interspersed with specially flagged nil elements. The operation of *packing* moves the elements of s, in their existing order, into the lower numbered processors and moves the nil elements into the higher numbered processors. Thus the packing operation can also be regarded as the operation of *segregating* unflagged from flagged elements by moving the former left and the latter right while maintaining the order of s (see Figure 3).

To pack, we first determine the destination processor $d(x)$ for each element x in s. This is simply the number of elements of s to the left of x and can be determined in time $O(\log N)$ by summing. We then interchange adjacent even/odd elements x, y according to the following rule:

> If x is in s and has an even location but an odd destination, then interchange x and y. Proceed similarly if x is in s and has an odd location but an even destination.

After all the above interchanges, each data item of s with an even (respectively, odd) destination will be in an even (respectively, odd) position. Moreover, each of these two groups of elements will be separately in order. We then apply packing in parallel to the odd and the even elements separately; of course, the effect of this is to pack s. The time required for packing is therefore $O(\log N)$. Note that to pack the even and odd elements separately, we first unshuffle, then separately pack the elements held in the lower and upper halves of our processor assemblage, then shuffle.

We can ensure that both s and its complement retain their original order simply by applying the above algorithm twice.

5.2 Sorting and Merging

The fundamental operation of sorting a collection of elements into ascending or descending order (using a parallel assemblage of computing elements) has been studied in several interesting papers: see Batcher [2], Knuth [15, pp. 232–233, 237], Nassimi and Sahni [19], Baudet and Stevenson [3]; see also Preparata [24], which gives a paracomputer rather than an ultracomputer algorithm, Valiant

Before packing:	A		B		C	D		E
	p0	p1	p2	p3	p4	p5	p6	p7
After packing:	A	B	C	D	E			

Fig. 3. Packing data to the left in an eight-processor ultracomputer. (Data items are indicated by uppercase letters, nil items by blanks.)

[28], and Hirschberg [14]. Batcher's *bitonic sorting* technique is a variant of the standard serial merge sort, but uses a nonobvious concurrent technique to accomplish merging. It is technically advantageous to think of the merge operation as transforming an array A whose top part is in increasing order and whose bottom part is in decreasing order into a fully sorted array. To avoid confusion, we refer to this somewhat nonstandard merging operation as a *murge*. Then bitonic sorting, like standard merge sorting, consists of the following steps:

(1S) sort the first half of the array into order and the second half into inverse order (this produces an array that increases up to its middle, then decreases);
(2S) then murge the two halves of the array.

The murge procedure can be defined recursively as follows: To murge an array of length $2N$,

(1M) compare each element x_j of the left half of the array with the corresponding element x_{j+N} of the right half of the array and interchange them if they are out of order;
(2M) then apply murging to the top and the bottom half of the array in parallel.

All the operations of the murge step (1M) can clearly be performed in parallel in time $O(1)$ using the shuffle connections. Thus the time $M(2N)$ needed to murge an array of $2N$ elements by the method indicated satisfies $M(2N) = M(N) + O(1)$, which means that $M(N) = O(\log N)$. Hence the time $S(N)$ needed to sort an array of elements bitonically satisfies $S(2^k) = S(2^{k-1}) + O(k)$, from which it follows that $S(N) = O(\log^2 N)$.

An arbitrary interelement comparison function can be used within a bitonic sort; thus we can sort into the order defined by any such function. Secondary elements associated with a primary sort key x can be moved with x during sorting; thus we can sort pairs by their first or second component, etc.

The murge operation can be used to realize ordinary merging of the two halves of an array A simply by reversing the second half of A and then murging. Hence ordinary merging can be accomplished in time $O(\log N)$. In view of the importance of sorting to many other operations, each of which is quite important in its own right, we write $\Sigma(N)$ for the time needed to sort N items on an N-processor ultracomputer. The existence of the bitonic algorithm shows that $\Sigma(N) = O(\log^2 N)$. Of course $\Sigma(N) = \Omega(\log N)$. It is not known if the bitonic algorithm is optimal.

Any fast sorting procedure can be used as the basis for an effective treatment of various important set-theoretic operations, including $s \cup t$ (set union), $s \cap t$ (set

intersection), $s - t$ (set difference), $f \circ g$ (composite map formation), and $f[s]$ (range of the map f on the set s).

In discussing the set-theoretic operations listed above, we first suppose that the number of elements of each set s to be considered is no greater than the number 2^D of processors in the ultracomputer, and accordingly that the elements of s are stored, one element per processor p, all in similarly numbered memory cells $p(k)$, where k is an "address" locating s.

Empty positions in the representation of a set can be filled with some distinguishable **nil** element. We regard maps f as sets of ordered pairs and store them in similar fashion: the first component of each pair in a cell $p_n(k)$, where p_n is the processor which holds the pair; the second component of the pair in cell $p_n(k + 1)$.

Occasionally we will need to assume that the elements of a set are held in some sorted order within the processors of an ultracomputer. In such cases we will not necessarily assume that the set elements are packed into a dense range of processors, but will allow **nil** elements to be interspersed among the set elements. Sometimes we may also wish to assume that the pairs constituting a set occur in sorted order of their first elements; here also we allow interspersed **nil** pairs.

To realize the set-theoretic operations in which we are interested, we proceed as follows:

(1) $s - t$, $s \cap t$. To implement set difference, we take the lower half of s together with the lower half of t and sort them together. Then we convert all duplicated elements of s and all elements of t to **nil**, thus obtaining the difference s_0 of the low half s_- of s with the low half t_- of t. Then, proceeding in much the same way, we form $s_0 - t_+$, where t_+ is the upper half of t; this gives $s_- - t$. Forming $s_+ - t$ in the same way, and then packing and joining these two halves, we get $s - t$. The intersection $s \cap t$ is $s - (s - t)$ but can also be formed more directly.

If the elements of s and t are initially sorted into the same order, then merging rather than sorting can be used in the procedure just described, allowing $s - t$ and $s \cap t$ to be formed in time $O(\log N)$. In general, this algorithm is $O(\Sigma(N))$.

(2) $s \cup t$. Assuming that $s \cup t$ has fewer than 2^D elements, we can form it by first forming $s - t$ and packing the latter to a low position, then packing t to a high position, joining $s - t$ and t, and packing this totality, which is $s \cup t$. Note that the union of two maps can be formed in much the same way. This general procedure forms $s \cup t$ in time $O(\Sigma(N))$; if s and t are sorted into the same order, time $O(\log N)$ is sufficient.

(3) $f \circ g$, i.e., $(f \circ g)(x) = f(g(x))$. If f is not single-valued, then the cardinality of $f \circ g$ can be as large as the product of the separate cardinalities of f and g. We prefer to avoid this issue, and hence we simplify by supposing that f is single-valued. We then proceed "by halves" as we did in the algorithm sketched under (1) above; this simplifies our discussion by reducing it to the case in which neither f nor g contains more than 2^{D-1} elements. Assuming this simplification, we proceed as follows:

(a) Mark the pairs of f to distinguish them from the pairs of g.

(After)								
Step (1)	(1, 2)	(2, 3)*	(3, 6)	(4, 2)	(5, 2)*	(3, 4)		(4, 5)*
Step (2)	(2, 3)*	(5, 2)*	(4, 5)*		(1, 2)	(3, 6)	(4, 2)	(3, 4)
Step (3)	(2, 3)*	(1, 2)	(4, 2)	(4, 5)*	(3, 4)	(5, 2)*	(3, 6)	
Step (4a)	(2, 3)*	(2, 3)* (1, 2)	(2, 3)* (4, 2)	(4, 5)*	(4, 5)* (3, 4)	(5, 2)*	(5, 2)* (3, 6)	(5, 2)*
Step (4b)	(2, 3)*	(2, 3)* (1, 3)	(2, 3)* (4, 3)	(4, 5)*	(4, 5)* (3, 5)	(5, 2)*	(5, 2)*	(5, 2)*
Step (5)	(1, 3)	(4, 3)	(3, 5)					

Fig. 4. Calculating a functional composition in an eight-processor ultracomputer.

(b) Use the packing algorithm to move the pairs of f into low position and the pairs of g into high position.

(c) (Stably) sort the two sets of pairs together, the f pairs by their first and the g pairs by their second component.

(d) (Now we wish to replace the second component y of each g pair $[x, y]$ by the unique second component of the first f pair lying to the left of $[x, y]$, if any such exists and has a first component matching the second component of $[x, y]$.) Use the taking primitive to propagate each marked pair $[y, z]^*$ to all following unmarked pairs (up to the next marked pair). Replace all unmarked pairs $[x, y]$ to which a marked $[y, z]^*$ has been propagated by $[x, z]$. Replace all $[x, y]$ to which a marked $[y', z']^*$ with $y' \neq y$ has been propagated by **nil**.

(e) Drop all marked pairs. Pack the remaining pairs, which represent the functional composition $f \circ g$.

Figure 4 illustrates this process.

The functional composition algorithm is dominated by the sorting time which appears in step 3c; thus it requires time $O(\Sigma(N))$. If f and g are both monotone and sorted into similar orders, functional composition can be accomplished in time $O(\log N)$ by using merging in place of sorting.

The range $f[s]$ of f on s is simply the range of $f \circ i_s$, where i_s is the identity map on s; hence $f[s]$ can also be formed in time $O(\Sigma(N))$ (or in time $O(\log N)$ if f and s are sorted into corresponding orders).

(4) Occasionally we will wish to form the Cartesian product $s \times t$ of two sets. Since in accordance with our general assumption we will want this Cartesian product to contain no more than 2^D elements, we suppose, in order to simplify, that neither s nor t contains more than $2^{D/2}$ elements. Assuming this simplification we can proceed as follows. If we think of the $N = 2^D$ successive processors of an ultracomputer as being arranged in $2^{D/2}$ elements each, it is easily seen that $D/2$ successive shuffle operations have exactly the effect of transposing data items in

this two-dimensional arrangement. Thus we can

(a) pack both s and t, which will put all the data into the first "row" of processors;

(b) "transpose" in the manner indicated, to put all the data into the first "column" of processors;

(c) use the taking operation described previously, thereby copying the data items representing s (or t) from the first processor in each row to all the processors of the row;

(d) transpose the duplicated t items.

This last step will clearly bring every pair (s_i, t_j) of original data items together in some processor of the two-dimensional array.

5.4 Interprocessor Communication in an Ultracomputer

We have already observed that even those processes (like finite-element solutions of a numerical continuum problem) which can run relatively independently on multiple processors will require some limited amount of interprocessor communication. The more the need for communication can be minimized, the more effectively concurrency can be exploited; but when interprocessor communication becomes inevitable, we wish to accomplish it efficiently enough so as not to be swamped by the communications requirements of a largely parallel computational process.

In this section we consider the following three communication problems:

(1) *Information Request.* Here each processor p_n stands ready to supply some specified data item $f(n)$ on request; moreover, each p_n requests the data which p_m, $m = r(n)$, supplies. Thus our problem is that of moving the data item $f(r(n))$ into p_n. To do this, we proceed as follows:

(a) Form the functional composition $f \circ r$ by the method explained in the previous section. This requires time $O(\Sigma(N))$ and yields a set of pairs $[n, x]$.

(b) We must now move each pair $[n, x]$ into the corresponding processor p_n. To do this, simply sort these pairs by their first component, which again requires time $O(\Sigma(N))$.

Next suppose the pattern r of request addresses in this communication problem is known (and can be analyzed) in advance (but that the data items $f(n)$ change dynamically). Then the Clos-Benes data permutation algorithm described in Section 5.6 below can be used to transmit the requested data in parallel in time $O(\log N)$. The essential fact here is that $f \circ r$ can be formed in time $\log N$. To do this, we proceed as follows. We set up a permutation q which maps each of the sets $\{j \mid r(j) = k\}$ into a contiguous sequence of processor identifiers. Then we mark the first element of each such sequence and let $a(j)$ be the element in the first marked position which lies at or to the left of j, so that $r = a \circ q$, and $f \circ r = f \circ a \circ q$. Since the Clos-Benes data permutation algorithm will be seen to form $f \circ a \circ q$ from $f \circ a$ in $\log N$ steps, our problem reduces to forming $f \circ a$ in time $O(\log N)$. This can be done easily by using another permutation and the taking primitive.

(2) *Message Delivery.* While particularly simple, the data communication problem that we have just discussed serves to illustrate the fact that any ultracomputer data communication problem can be regarded as the problem of forming an appropriate map composition and hence can be reduced to one or more applications of the method described in step 3 of Section 5.3. As a second illustration of this general point, we consider the problem of *message delivery.* Here we suppose that each processor p_n generates a destination address $d(n)$ (which must be the identifier of some other processor), attaches a *message* $f(n)$ to this destination address, and adds its own identifier n as a *return address.* We then aim to deliver exactly one message to each destination address. Since only one message can be delivered to a given destination address on each communication cycle, even though several messages might be addressed to a given destination, we must treat all but one of the messages to a given destination as (temporarily) undeliverable. We shall then insist that our communication mechanism return a notification of nondelivery to the sender of each undelivered message.

To relate the communication operation that we have just described to the map primitives of step 3 of Section 5.3, let μd be the one-to-one submap of d such that $\mu d(n)$ is undefined whenever there exists an $n_0 < n$ such that $d(n_0) = d(n)$. The set of processors whose messages will be delivered during any given cycle are those which belong to the domain of μd. The message to be delivered to processor i is $g(i) = (f \circ (\mu d)^{-1})(i)$. We can therefore regard our communication problem as that of forming $f \circ (\mu d)^{-1}$. Notification of nondelivery should go to processor p_i if $(f \circ (\mu d)^{-1} \circ d)(i) \neq f(i)$. This shows that communication can be accomplished in time proportional to $\Sigma(N)$.

(3) *Party-Line Communication.* In party-line communication we assume that the processors of our assemblage are grouped into parties by a function g which maps each processor identifier into a group identifier. Each processor that wishes to transmit during a given cycle originates a message f. One transmitting processor p_n of each group is then selected arbitrarily, and its message $f(n)$ is sent to all the other elements of the group.

To express this communication problem in terms of map operations, let $f\phi g$ map each group identifier γ into the identifier m of the smallest processor p_m in the group γ for which $f(n)$ is defined. Then $f \circ (f\phi g) \circ g$ defines the pattern of information we wish to communicate. To form $f\phi g$, we can proceed as follows:

(a) The pairs $[n, \gamma]$ of γ and $[n, f(n)]$ of f will lie together in processor p_n. Replace all pairs $[n, \gamma]$ for which $f(n)$ is undefined by specially marked pairs $[n, \gamma]^*$. This requires time $O(1)$.

(b) (Stably) sort all the resulting pairs $[n, \gamma]$ and $[n, \gamma]^*$ on their second component, putting pairs $[n, \gamma]^*$ after pairs $[n, \gamma]$ with the same second component. This requires time $O(\Sigma(N))$.

(c) Use the taking primitive to transmit the first index n_0 in a contiguous group of pairs $[n, \gamma]$, all sharing the same second component, to all the other pairs of the group, and form the pairs $[n, n_0]$. This requires time $O(\log N)$ and forms a set of pairs representing $f\phi g$.

This makes it plain that communication can be accomplished in time $O(\Sigma(N))$.

5.5 A Remark Concerning Ultracomputer/Paracomputer Relationships

In Section 2 we defined the notion of *paracomputer*, that is, a parallel assemblage of processors in which all communication operations require only one cycle. The existence of the communication primitives which we have just described makes it plain that a paracomputer communication cycle can be emulated by an ultracomputer in time $O(\log N)$ (if the required pattern of accesses is static and known in advance), or in time $O(\Sigma(N))$ (even if the pattern of accesses varies dynamically in completely general fashion). In fortunate cases the access pattern used by a paracomputer algorithm will be close enough to the shuffle sequences available in an ultracomputer for the paracomputer algorithm to convert easily into an ultracomputer algorithm of the same asymptotic efficiency. Thus, when translated for ultracomputers, paracomputer algorithms can fall into any one of the following classes.

(i) *Communication costs adaptable to shuffle pattern*: No loss of asymptotic efficiency.
(ii) *Fixed communication pattern, not adaptable to successive shuffling*: Asymptotic efficiency reduced by factor $\log N$.
(iii) *Dynamic communication pattern*: Asymptotic efficiency reduced by factor $\Sigma(N)$.

5.6 Additional Details Concerning the $O(\log N)$ Permutation Algorithm

Several times thus far we have used the fundamental fact, which goes back to Clos [6] and Benes [4], that an arbitrary permutation f can be realized on a shuffle machine in time $O(\log N)$. See Waksman [29] for a proof of this fact using Hall's combinatorial "matching" or "marriage" theorem. It is useful to construct the required sequence of shuffles and interchanges in a manner independent of Hall's theorem, which we can do as follows. Let f be the permutation of processor identifiers to be realized. We say that f *preserves halves* if it maps each of the two sets $H_0^{1/2} = \{0 \cdots 2^{D-1} - 1\}$ and $H_1^{1/2} = \{2^{D-1} \cdots 2^D - 1\}$ into itself; that it *preserves quarters* if it maps each of the four sets $H_J^{1/4} = \{J \cdot 2^{D-2}, J \cdot 2^{D-2} + 1, \ldots, (J+1)2^{D-2} - 1\}$, where $J = 0, 1, 2, 3$, into itself, etc. We say that a permutation g *preserves pairs*, or is an *interchange map*, if each of the pairs $\{2n, 2n+1\}$ is mapped into itself (i.e., either interchanged or not interchanged) by g.

Let σ denote the "perfect shuffle" permutation, and let f be an arbitrary permutation. Our key aim is to show that there exist two pair-preserving permutations $g = G(f)$, $\hat{g} = \hat{G}(f)$ such that

$$\hat{f} = \sigma^{-1} g f \hat{g} \sigma \tag{7}$$

preserves halves. Moreover, if f itself preserves halves (or quarters, ..., etc.), then $\hat{f} = \hat{F}(f)$ will preserve quarters (or eighths, ..., etc.). It follows from this that an arbitrary permutation f can be factored into a product of shuffles, inverse shuffles, and interchange maps of the form

$$f = g_1 \sigma g_2 \cdots g_{D-1} \sigma \bar{g}_D \sigma^{-1} \hat{g}_{D-1} \cdots \sigma^{-1} \hat{g}_1, \tag{8}$$

where as usual $D = \log N$, and where the interchanges g_j, $\hat{g}_j$, $\bar{g}_D$ appearing in (8)

have the following inductive definition:

$$\begin{aligned} &f_1 = f; \qquad f_{j+1} = \hat{F}(f_j) \qquad \text{(cf. (7) above)}; \\ &g_j = G(f_j); \qquad \hat{g}_j = \hat{G}(f_j); \qquad \bar{g}_D = f_D. \end{aligned} \tag{9}$$

Thus, *once the necessary interchanges g_j, $\hat{g}_j$, $\bar{g}$ have been precalculated using* (9), the permutation f can be realized in $O(\log N)$ shuffles and interchanges.

To show how to construct the basic factorization (7), we note that σ^{-1} maps the set E of even integers (respectively, the set O of odd integers) onto the lower (respectively, upper) half of the range $\{0 \cdots 2^D - 1\}$. Therefore we need to construct pair-preserving g, $\hat{g}$ such that $\bar{f} = gf\hat{g}$ maps each of E and O onto itself.

To do this, we first let η be the map which interchanges all pairs of adjacent even-odd integers, so that $\eta(2n) = 2n + 1$, $\eta(2n + 1) = 2n$; note that $\eta(\eta(n)) \equiv n$. Next we aim to find a set E^* which, like the even numbers, includes exactly one member of each even-odd pair and which is such that $f[E^*]$ also includes exactly one member of each even-odd pair. (Here and below we use $A[B]$ to denote the range of the map A on the set B.) Assume for the moment that E^* has been constructed. Let $\hat{g}$ (respectively, g) interchange $(2n, 2n + 1)$ iff E^* (respectively, $f[E^*]$) contains $2n + 1$ rather than $2n$. Then clearly $gf\hat{g}[E] = gf[E^*] = g[E^*] = E$, as required.

To construct the set E^*, put $\eta f^{-1}\eta f = f_1$, so that $f_1^{-1} = \eta f_1 \eta$. Then we can never have $f_1^k(n) = \eta(n)$, since if k were the smallest integer for which this happened, we would have $f_1^{k-1}(n) = \eta f_1(n)$; thus $f_1^{k-2}(m) = \eta(m)$ for $m = f_1(n)$, a contradiction unless $k = 1$. But $f_1(n) = \eta(n)$ implies $\eta f(n) = f(n)$, which is also impossible. From this it follows that the set $\{f_j^k(n), -\infty < k < \infty\}$ never includes both of any even-odd pair of integers; hence if $\eta f_1^k(n) = f_1^j(n)$, we would have $f_1^{-k}\eta(n) = f_1^j(n)$, so $\eta(n) = f_1^{j+k}(n)$, contradicting what has just been proved. Therefore, if we let $\mu(n)$ be the minimum of all $k_1^k(n)$, $-\infty < k < \infty$, and put $E^* = \{n \mid \mu(n) \text{ is even}\}$, $O^* = \{n \mid \eta(n) \text{ is odd}\}$, then E^* and O^* are disjoint sets. From the fact that $\eta f_1 \eta = f_1^{-1}$ it follows that each of E^* and O^* contains exactly one of every even-odd pair, and hence $E^* = \eta[O^*]$. Moreover, $f_1[E^*] = E^*$ and $f_1[O^*] = O^*$. Then clearly $\eta f^{-1}\eta f[E^*] = E^*$, so $\eta f[E^*] = f\eta[E^*] = f[O^*]$. Thus $f[E^*]$ and $\eta f[E^*]$ are disjoint, so that $f[E^*]$ includes exactly one member of each even-odd pair, as required.

This shows the existence of the factorization (8), which enables us to realize an arbitrary data permutation in time $O(\log N)$. Moreover, it gives us a recipe for calculating the interchange mappings g_j, $\hat{g}_j$ which appear in this factorization. By what has just been said, computation of these mappings is as expensive as a $(\log N)$-fold repetition of the following step: Given a permutation h of the integers $0, \ldots, 2^D - 1$, calculate the function $\mu(n) = \min_k h^k(n)$.

This function can be calculated as follows. Let $a_0(n) = n$ and $b_0(n) = h(n)$. Then inductively, let $a_{j+1}(n) = \min(a_j(n), a_j \circ b_j(n))$ and $b_{j+1} = b_j^2$. It follows that $b_j = h^{2^j}$, and

$$a_j(n) = \min(n, h(n), h^2(n), h^3(n), \ldots, h^{2^j-1}(n));$$

hence $\mu(n) = a_D(n)$. Thus calculation of $\mu(n)$ requires time $O(\Sigma(N) \log N)$. It follows that precalculation of all the interchange permutations appearing in (8)

above can be accomplished in time $O(\Sigma(N) \log^2 N)$. Since data can be transferred by sorting in time $O(\Sigma(N))$, it follows that precalculation of the factorization (8) prior to an extended communication operation in which a fixed permutation pattern is used repeatedly is advantageous if the number of words transferred per processor is $\Omega(\log^2 N)$.

6. OTHER ULTRACOMPUTER ALGORITHMS

We now review several of the more interesting ultracomputer algorithms listed in Table I above, starting with a parallel variant of Strassen's algorithm for matrix multiplication.

6.1 Matrix Multiplication

The largest square matrix that can be stored in an ultracomputer of size N without crowding more than one matrix element into any processor is an $N^{1/2} \times N^{1/2}$ matrix. To multiply such matrices, we can decompose them into $N^{1/3} \times N^{1/3}$ sections. The product of submatrices a_{ij} and b_{jk} of this smaller size can be formed by diffusing all the a_{ij} and all the b_{jk} to all positions (i, j, k), multiplying in parallel, and then summing. This will multiply two $N^{1/3} \times N^{1/3}$ sections in time proportional to log N. An $N^{1/2} \times N^{1/2}$ matrix can be partitioned as an $N^{1/6} \times N^{1/6}$ matrix of $N^{1/3} \times N^{1/3}$ sections; then two such matrices can be multiplied in time $O(N^{(\log_2 7)/6} \log N)$ using Strassen's method.

The preceding algorithm directly parallelizes the "naive" matrix multiplication algorithm which simply uses the definition $(ab)_{ij} = \sum a_{ik} b_{kj}$ of matrix multiplication. As pointed out by Chandra [5], an improved algorithm, which permits larger matrices to be multiplied in time close to log N, can be derived by adapting Strassen's technique. We shall give an ultracomputer adaptation of Chandra's paracomputer algorithm. To this end, suppose that $7^\alpha \le N$, and consider a pair of $2^\alpha \times 2^\alpha$ matrices a, b, whose components we write as

$$a_{i_1 j_1 \cdots i_\alpha j_\alpha} \qquad \text{and} \qquad b_{i_1 j_1 \cdots i_\alpha j_\alpha},$$

respectively, where each i_m and j_m ranges over the set $\{0, 1\}$. In the case $\alpha = 1$, Strassen's well-known identity has the form

$$(ab)_{i'j'} = \sum_m \Gamma_{i'j'm} \left(\sum_{ij} L_{mij} a_{ij} \right) \left(\sum_{ij} \Lambda_{mij} b_{ij} \right);$$

all the coefficients Γ, L, and Λ are either 0, 1, or -1, and $1 \le m \le 7$ (cf. Aho, Hopcroft, and Ullman [1, p. 231]). For the general case it follows that if we put

$$(La)_{m_1 \cdots m_\alpha} = \sum_{\substack{i_1 \cdots i_\alpha \\ j_1 \cdots j_\alpha}} L_{m_1 i_1 j_1} \cdots L_{m_\alpha i_\alpha j_\alpha} a_{i_1 j_1 \cdots i_\alpha j_\alpha},$$

$$(\Lambda b)_{m_1 \cdots m_\alpha} = \sum_{\substack{i_1 \cdots i_\alpha \\ j_1 \cdots j_\alpha}} \Lambda_{m_1 i_1 j_1} \cdots \Lambda_{m_\alpha i_\alpha j_\alpha} b_{i_1 j_1 \cdots i_\alpha j_\alpha}, \tag{10}$$

$$c_{i_1 j_1 \cdots i_\alpha j_\alpha} = \sum_{m_1 \cdots m_\alpha} \Gamma_{i_1 j_1 m_1} \cdots \Gamma_{i_\alpha j_\alpha m_\alpha} (La)_{m_1 \cdots m_\alpha} (\Lambda b)_{m_1 \cdots m_\alpha}.$$

Then $c = ab$.

In these formulas each m_j ranges over the set $1 \le m_j \le 7$. Suppose now that the matrix element $a_{i_1 j_1 \cdots i_\alpha j_\alpha}$ is initially held in processor p_n, where $n = j_\alpha i_\alpha \cdots j_1 i_1$.

Let $\beta < \alpha$ be an integer to be chosen later. By applying an appropriate data permutation, we can move $a_{i_1 j_1 \cdots i_\alpha j_\alpha}$ into $p_{\tilde{n}}$, where $\tilde{n} = i_1 j_1 \cdots i_\beta j_\beta\ 0\ i_{\beta+1} j_{\beta+1}\ 0\ i_{\beta+2}\ j_{\beta+2} \cdots 0\ i_\alpha j_\alpha$. Once this has been done, the elements $\sum_{i_\alpha, j_\alpha} L_{m_\alpha i_\alpha j_\alpha} a_{i_1 j_1 \cdots i_\alpha j_\alpha}$ can be formed in parallel and stored in the processors

$$p_{i_1 j_1 \cdots i_\beta j_\beta 0 i_{\beta+1} j_{\beta+1} \cdots 0 i_{\alpha-1} j_{\alpha-1} k_\alpha i_\alpha j_\alpha},$$

where k ranges over $\{0, 1\}$, but where the combination $(k_\alpha, i_\alpha, j_\alpha) = (0, 0, 0)$ is never used. The remaining seven values of $(k_\alpha, i_\alpha, j_\alpha)$ correspond to the seven values of m_α. Then, after applying the cube σ^3 of the shuffle map σ to the resulting data, we can form

$$\sum_{\substack{i_{\alpha-1}, i_\alpha \\ j_{\alpha-1}, j_\alpha}} L_{m_{\alpha-1} i_{\alpha-1} j_{\alpha-1}} L_{m_\alpha i_\alpha j_\alpha} a_{i_1 j_1 \cdots i_{\alpha-1} j_{\alpha-1} i_\alpha j_\alpha},$$

and so on iteratively until

$$q_{i_1 j_1 \cdots i_\beta j_\beta m_{\beta+1} \cdots m_\alpha} = \sum_{\substack{i_{\beta+1}, \cdots, i_\alpha \\ j_{\beta+1}, \cdots, j_\alpha}} L_{m_{\beta+1} i_{\beta+1} j_{\beta+1}} \cdots L_{m_\alpha i_\alpha j_\alpha} a_{i_1 j_1 \cdots i_{\beta+1} j_{\beta+1} \cdots i_\alpha j_\alpha} \tag{11}$$

has been formed. We choose β to be the smallest integer for which $8^{\alpha-\beta} \cdot 4^\beta \leq N$. Since $7^\alpha \leq N$, this restriction is milder than $(\frac{8}{7})^\alpha \leq 2^\beta$, so that $\beta \sim r\alpha$, where $r = \log(\frac{8}{7})/\log 2 < 1$.

Once the quantities (11) have been formed, we permute to put $q_{i_1 j_1 \cdots i_\beta j_\beta m_{\beta+1} \cdots m_\alpha}$ into processor $p_{M i_1 j_1 \cdots i_\beta j_\beta}$, where M is the index of the subscript $m_{\beta+1} \cdots m_\alpha$ in the set of all $7^{\alpha-\beta}$ such subscripts, and then apply the same procedure to form

$$\hat{q}_{i_1 j_1 \cdots i_\gamma j_\gamma m_{\gamma+1} \cdots m_\alpha} = \sum_{\substack{i_{\gamma+1}, \cdots, i_\alpha \\ j_{\gamma+1}, \cdots, j_\alpha}} L_{m_{\gamma+1} i_{\gamma+1} j_{\gamma+1}} \cdots L_{m_\alpha i_\alpha j_\alpha} a_{i_1 j_1 \cdots i_{\gamma+1} j_{\gamma+1} \cdots i_\alpha j_\alpha}. \tag{12}$$

We choose γ to be the smallest integer for which $4^\gamma \cdot 7^{\alpha-\beta} 8^{\beta-\gamma} \leq N$, which is a milder restriction than $4^\gamma \cdot 8^{\beta-\gamma} \leq 7^\beta$, and leads to $\gamma \sim r\beta$, with r as above. This process leads after $O(\log D)$ iterations to a permutation of La (cf. (7)). The time required to form these quantities is $O(\log N \log\log N)$. The quantities $(\Lambda b)_{m_1 \cdots m_\alpha}$ can be formed in the same way and stored in the same permuted arrangement. Then we can multiply these sets of quantities together in constant time. From the resulting set of products, the coefficients $c_{i_1 j_1 \cdots i_\alpha j_\alpha}$ can be formed in time like that required to form the quantities La and Λb and by a very similar technique. The technique just described multiplies two matrices of size $2^\alpha \times 2^\alpha$ in time $O(\log N \log\log N)$.

To keep the amount of local work done by the matrix multiplication algorithm in balance with the amount of communication it performs, it is appropriate to let each of the coefficients $a_{i_1 j_1 \cdots i_\alpha j_\alpha}$ and $b_{i_1 j_1 \cdots i_\alpha j_\alpha}$ be a $p \times p$ matrix. As usual, let $D = \log N$. Then each matrix entry consists of P^2 numbers, so to form La and Λb takes $O(P^2 D \log D)$ cycles. Moreover, to multiply the $P \times P$ coefficient matrices of La and Λb will take $O(P^{2.81})$ cycles, and finally, to form the c's from these products will take $O(P^2 D \log D)$ cycles. Thus two matrices of size $PN^{1/2.81}$ can be multiplied in time $O(P^2 D \log D + P^{2.81})$. This shows that Strassen's matrix multiplication algorithm is completely parallelizable at level $(\log N \log\log N)^{1/0.81}$.

6.2 Matrix Inversion

In view of the central importance of Gaussian elimination and matrix inversion for numerical analysis, it would be quite useful to have a good parallel matrix inversion algorithm. In the case of subdiagonal matrices, that is, a_{ij} for which $a_{ij} = 0$ if $j > i$, this is easy. Assume that we deal with $n \times n$ matrices where $n^{2.81} \leq N$. Then we can form, $a_{ii}^{-1}a_{ij}$ in time $O(\log N)$, and hence suppose that $a = I - b$, where $b_{ij} = 0$ if $j \geq i$. Then using the identity $(I - b)^{-1} = (I + b)(I + b^2)(1 + b^4) \cdots$ (and noting that $b^n = 0$), inversion is accomplished in time $O(\log^2 N \log\log N)$ (see Orcutt [21] and Sameh and Brent [26]).

An algorithm for inverting more general matrices can be developed along the lines of Csanky [7], as improved by Preparata and Sarwate [25]. (Note, however, that the technique to be explained is hopelessly unstable from the numerical point of view; thus it can only be used in rational or modular arithmetic calculations.) The idea here is to find the coefficients of the characteristic polynomial $p(x) = x^n + c_{n-1}x^{n-1} + \cdots + c_0$ of the $n \times n$ matrix b to be inverted, and then to use the relationship $b^{-1} = -c_0^{-1}(b^{n-1} + c_{n-1}b^{n-2} + \cdots + c_1)$. To this end, we proceed as follows. Suppose for simplicity that $n = 4^k$, and let $m = n^{1/2}$. From b form b^2, then from b and b^2 form b^3 and b^4, then b^5, b^6, b^7, b^8, etc., until all powers of b up to b^m have been formed. This requires time $O(\log^2 N \log\log N)$, but in order that all the necessary multiplications can proceed in parallel, we require $n^{2.81} \cdot m = n^{3.31} \leq N$.

Once the matrices b^j, $1 \leq j \leq m$, have been formed, form all the matrices $b^{j \cdot m}$. This can be done in time $O(\log^2 N \log\log N)$ by repeated squarings in parallel, but again we require that $n^{2.81} \cdot m = n^{3.31} \leq N$. Next proceed in parallel to form all the traces $\mathrm{tr}(b^{j_1}b^{j_2 \cdot m})$. These are just inner products and can all be formed in parallel in time $O(\log N)$ using the permutation and summation primitives, since $n^2 \cdot m \cdot m = n^3 < N$. This gives us all the quantities $s_j = \sum_{i=1}^{n} \lambda_i^j$, $j = 0, \ldots, n$, where $\{\lambda_i\}$ is the sequence of eigenvalues of b. The coefficients c_j of the characteristic polynomial of b can be calculated rapidly from the quantities s_j by making use of a classical relationship which is described below. Suppose for the moment that these coefficients are available. Then we can form all the sums

$$d_i = \sum_{j=0}^{m-1} c_{j+im}b^j, \qquad i = 0, \ldots, m-1, \tag{13}$$

in parallel in time $O(\log N)$, since the matrices b^j, $j = 1, \ldots, m$, are available. Clearly $\sum_{i=0}^{m-1} d_i b^{im} = \sum_{j=0}^{n-1} c_j b^j = -c_0 b^{-1}$. The m product matrices $d_i b^{im}$ can all be formed in parallel in time $O(\log N \log\log N)$, since all the b^{im} are already available and $n^{2.81} \cdot m = n^{3.31} \leq N$. Thus b^{-1} can be formed in time $O(\log^2 N \log\log N)$.

To calculate the coefficients c_j from the s_j, we have only to use the "Newton identity" or "Leverrier relationship" (see Csanky [7] and Preparata and Sarwate [25]),

$$\begin{bmatrix} 1 & & & & \\ s_1 & 2 & & & \\ s_2 & s_1 & 3 & & \\ \vdots & \vdots & \ddots & \ddots & \\ s_{n-1} & s_{n-2} & \cdots & s_1 & n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = - \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix}. \tag{14}$$

Since the matrix appearing in (14) is lower triangular, it can be inverted in time $O(\log^2 N \log\log N)$.

6.3 Miscellaneous Combinatorial Algorithms

(1) *Map Analysis Problems.* Suppose that g is a map of the set of integers $0, \ldots, 2^D - 1$ into itself, and that $\mu(n) = \min_k g^k(n)$ as in Section 5.6. Consider the problem of finding $\nu(n) = \min\{k \mid g^k(n) = \mu(n)\}$. This can be handled as follows. Calculate μ, and put $\nu(n) = 0$ for all n such that $n = \mu(n)$. For $j = 1, \ldots, D$, put $b_j = g^{2^j}$, and put $\nu(n) = \nu(b_j(n)) + 2^j$ for all n for which $\nu(n)$ has not yet been defined and for which $\nu(b_j(n))$ is defined. Then $\nu(n)$ will be fully defined as soon as j reaches D. The time required for this calculation is $O(\Sigma(n) \log N)$.

If g is a permutation, this gives us the cycle decomposition of g in $O(\Sigma(N) \log N)$ steps. If g can be many-to-one, one will sometimes want to calculate the intersection of the ranges of all the powers g^j. This is simply the range of g^{2^D} so that it can also be calculated in $O(\Sigma(N) \log N)$ steps.

(2) *Connected Components of an Unordered Graph.* Suppose that a graph G is defined by its edges, given as a set of pairs $[x, y]$ stored one pair per processor, and that we wish to form the set of connected components of G. We can easily enumerate the nodes of G by sorting its edges; hence we shall suppose that the nodes of G are integers not exceeding $2N$. We will represent the connected components of G by constructing a map μ_G which sends each node n into a representative node belonging to the same component as n. To do this, we let K be the set of all nodes n upon which no edge other than $[n, n]$ is incident, and let L (respectively, H) be the set of nodes n which appear in an edge $[n, n']$ with $n' > n$ (respectively, $n' < n$). If $\#L \geq \#H$, let f map each $n \in L$ into the largest n' for which $[n, n'] \in G$, and if $\#L < \#H$, let f map each $n \in H$ into the smallest n' for which $[n, n'] \in G$. The sets K, L, H and the map f can be constructed in time $O(\Sigma(N))$. If $\#L \geq \#H$ (respectively, $\#L < \#H$), let $g(x)$ with domain L (respectively, H) and range $H - L$ (respectively, $L - H$) be the highest power $f^m(x)$ which is defined. The map g can be formed from f by repeated squarings in time $O(\Sigma(N) \log N)$. Let G' be the set of all edges $[g(n), g(n')]$ for $[n, n'] \in G$ and $n \neq n'$. Then G' is a graph with no more than half as many nodes as G. Then use this same procedure recursively to form $\mu_{G'}$. We have $\mu_G(n) = n$ if $n \in K$, $\mu_G(n) = \mu_{G'}(g(n))$ if $n \in L$ (respectively, H), $\mu_G(n) = \mu_{G'}(n)$ if $n \in H - L$ (respectively, $L - H$). The time required for this algorithm is clearly no more than $O(\Sigma(N) \log^2 N)$. A rather similar construction, requiring similar asymptotic time, will form a spanning forest for G (see also Hirschberg [13] and Levitt and Kautz [17]).

(3) *Tree Summation, Preorder, Postorder.* Let a tree t be defined by its father map f, which we suppose to be given as a set of pairs $[x, y]$ stored one pair per processor. Suppose that a numerical weight $w(x)$ defined on the nodes x of t is represented in the same way. Occasionally one will want to form the sum $a(x)$ of $w(y)$ over all the ancestors y of x, or the sum $d(x)$ of $w(y)$ over all the descendants y of x. To obtain $a(x)$, form the powers f^{2^j} and simultaneously the sum $a_j(x)$ of $w(y)$ over the 2^j nearest ancestors of x; then $a_{j+1}(x) = a_j(x) + a_j(f^{2^j}(x))$. To obtain $d(x)$, form the powers f^{2^j} and simultaneously the sum $d_j(x)$ of $d(y)$ over all descendants y of x whose tree distance from x is less than 2^j; then calculate $d_{j+1}(x) = d_j(x) + \sum_{y \in s_j(x)} d_j(y)$ using summation by groups and sorting, where $s_j(x) = (f^{2^j})^{-1}(x)$. Both a and d can be formed in time $O(\Sigma(N) \log N)$.

Various more complex sums over the nodes of a tree can be formed using the procedure just described. Suppose, for example, that the tree t is ordered and that we wish to form the sum $c(x)$ of $w(y)$ over those ancestors y of x for which the tree path from y to x begins with the rightmost immediate descendant of y. To do this, we can first define $\hat{w}(x) = w(f(x))$ if x is the rightmost descendant of $f(x)$; otherwise $\hat{w}(x) = 0$. Then let $\hat{a}(x)$ be the sum of $\hat{w}(y)$ over the ancestors y of x, and finally put $c(x) = \hat{w}(x) + \hat{a}(x)$.

A tree t represented by its father map f is unordered but can be ordered by assigning any node order and ordering the set of descendants of any node n in this order. The father map f in the binary tree t' corresponding to the ordered tree t is then $f'(n) = f(n)$ if n has no smaller sibling; otherwise $f'(n) =$ next smaller sibling. f' can be calculated in time $O(\Sigma(N))$ by sorting. Let λ denote the left descendant of a node x. The postorder number $\pi_+(n)$ of each node n in t' can be calculated as $\delta(n) + \sum_m \delta(\lambda(m))$, where $\delta(n)$ is one plus the number of descendants of n in t' and m ranges over all those ancestors of m in the binary tree t' of which n is a rightmost t'-descendant. The preorder number $\pi_-(n)$ of each node n in t' can be calculated as $\sum_m \delta(\lambda(m)) + \alpha(n) + 1$, where α is the number of ancestors of n in t', and where δ and the range of m are as before. This shows that both the preorder and the postorder numbers of the nodes of an ordered tree t can be calculated in time $O(\Sigma(n) \log N)$.

(4) *Backtracking Search.* A backtracking search program is one which explores a tree of possibilities by selecting branches for exploration in a manner which can be nondeterministic, but retains a copy of alternate data environments to backtrack to if and when a current path of exploration fails. In using an ultracomputer to carry out such a search, one would aim to keep all processors busy by sending environments to be explored from processors currently busy with other branches to processors with nothing to do. This might be organized as follows. Processors with surplus environments would occasionally insert their identifiers into a set A, and idle processors would insert their identifiers into another set B. Then by packing we would match elements of A with elements of B. A processor whose identifier was matched to an element p of B would then be notified as to the identity of p and would send one of its surplus environments to p, which would begin to explore this environment.

7. ALTERNATIVE ARCHITECTURES AND PHYSICAL STRUCTURES

7.1 Shuffle Powers and Nearest Neighbor Communication

Certain important problems have a latticelike geometric structure in two, three, or four dimensions. For problems of this kind, particularly rapid parallel communication between nearest lattice neighbors can be advantageous. For this reason it may be desirable to provide connections which realize powers σ^k, σ^{-k} of the shuffle map directly. Then the number of cycles necessary to access $p_{n\pm 2^j}$ from p_n is equal to $1 + 2d$, where d is the number of steps $i \to i \pm 1$, $i \to i \pm k$ needed to reach j. (The accessing sequence involves an appropriate product of d shuffles $\sigma^{\pm 1}$ and shuffle powers $\sigma^{\pm k}$, a set of load operations performed in parallel in one cycle, and then an inverse sequence of shuffle operations.) Suppose, for example, that $D = 14$. Then by providing both σ and σ^4 as one-cycle operations,

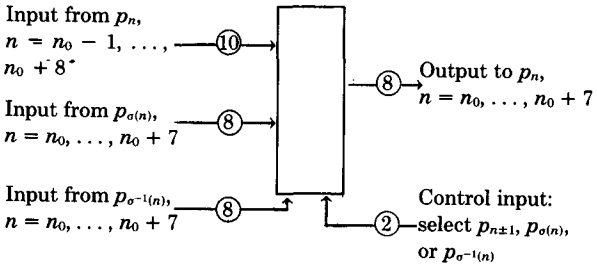

Fig. 5. Data routing chip.

communication between p_n and $p_{n\pm2^j}$ is possible in the number of cycles shown in the following table:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
cycles	1	3	5	5	3	5	5	7	5	5	3	5	5	3

A 16K-processor ultracomputer with these connections can be used efficiently either as a 128 × 128-processor space array, a 32 × 32 × 16 solid array, or a 16 × 16 × 8 × 8 hypersolid array.

7.2 Physical Layout for an Ultracomputer

Consider a (mini!) ultracomputer consisting of 16K 16-bit processors, each consisting of one processing chip and two memory chips (possibly 8 × 132K memory chips, 40 pins; chips of this sort should become available within the next decade). A total of 48K processing and memory chips are therefore required. To realize the shuffle connection, we can use data routing chips with the inputs and outputs shown in Figure 5. Since each such chip handles eight processors, we would need 2K of them for each line provided between processors.

The next issue to consider is packaging. Assume that the chips will be mounted on $1\frac{1}{2} \times 1\frac{1}{2}$-foot multilayer boards, both sides, 512 chips per board, and that 32 boards (16K chips total) will be held in $1\frac{1}{2} \times 2 \times 2$-foot racks with backplanes, two of which combined will constitute one 3 × 2 × 2-foot cabinet of 32K chips. This would allow us to use eight connections between processors (32K connecting chips) in a total three-cabinet 3 × 4 × 4-foot configuration. A 20-ns communication cycle could then be used. A full 16-bit-wide data permutation operation might then require approximately 60 cycles, or 1.2 μs.

The assumed configuration would require about 64K chips, and be manufacturable for about one million dollars, given the technology that we have assumed.

7.3 Wireability Considerations

Each 512-chip board would hold approximately 128 processors (384 chips), plus associated switching chips. Assuming that processors will be grouped on boards in such a way as to minimize the number of wires which come off each board, we can estimate that 16 wires per processor will come into each board, for a total of approximately 2400 wires per board. These can be led to the four edges of the board, 500 wires per edge, or approximately 30 connections per inch. Approximately half the wires coming from each board of a cage will go to other boards in

the same cage; thus approximately 32,000 wires will enter each cage, about one-quarter of which should come from the other cage in the same cabinet. This means that about 150,000 wires must be handled by intercabinet connections.

If handling this many connections proves to be impossible, we can timeslice the connections. This may reduce our attainable computation rate significantly unless very high bandwidth connections (possibly fiber optics) are used, but could allow the number of intercabinet connections to be as low as 10,000.

7.4 A Layered Architecture

The most problematic aspect of the ultracomputer architecture suggested in the preceding section would appear to be the very large number of intercabinet wires which it implies. For this reason we will now consider an alternative, somewhat less efficient architecture in which a 16K processor ultracomputer is built up out of 16 "layers," each of which is itself a 1024-processor ultracomputer. Designate the kth processor of the jth layer as $p_{k,j}$. Then we suppose that the following connections are provided: $p_{k,j} \leftrightarrow p_{\sigma(k),j}$, where σ is the shuffle map appropriate to a 1,024 processor ultracomputer; $p_{k,j} \leftrightarrow p_{k+1(\text{mod } 1,024),j}$; and $p_{k,j} \leftrightarrow p_{k,j+1(\text{mod } 16)}$. It will be seen that this leads to a more regular and manageable wiring pattern and to only a small performance degradation.

To see that the proposed interconnections degrade performance only slightly, we need to describe ways in which the fundamental permutation, summing, packing, and sorting algorithms can be performed efficiently.

(1) *Permutation.* To permute data (in a known pattern), we first permute the data in each layer in such a way that after this permutation no two processors $p_{k,j}$ with the same k contain data with the same destination layer; then we move each data item to its destination layer; then we permute the data in each layer once more, to move each data item to its final destination. To see that there always exists a permutation of each layer which ensures that after the permutation no two processors $p_{k,j}$ with the same k contain data with the same destination layer, we can argue as follows. For each j, let D_j be the set of destination layers of data items held in the processors $p_{k,j}$. If J is any set of layers, then $\#(\bigcup_{j\in J} D_j) \geq \#J$, since otherwise all the $M \cdot \#J$ data items in the layers of J would have only $M \cdot (\#J - 1)$ destinations, where M is the number of k we are considering. It follows by Hall's "marriage" or "matching" theorem that we can select a data item from each layer in such a way that all these items have different destination layers. Permuting each layer so as to put these items into the processors $p_{1,j}$, and continuing inductively, we define the desired permutation of each layer.

To move items to their destination layers, we can use the $p_{k,j} \leftrightarrow p_{k,j+1}$ connections to shift elements circularly between layers up to 15 times. This requires 15 cycles; the remainder of the permutation operation requires 37 cycles, for a total of 52 cycles in all.

(2) *Summing.* To sum, we first sum separately in each layer; then proceeding serially for 15 cycles, we form the sequence of sums $\sum_{i<j} t_i$, where t_i is the sum of all the data items in the ith layer, and then sum once more by layers to form the desired partial sums.

(3) *Packing.* We first sum to determine the destination processor $p_{k,j}$ for each data item, then use a modified packing procedure in each layer to position each data item into a processor in the same layer but with appropriate k, and then

move the data item in $p_{k,j}$ to the appropriate $p_{k,f}$. This last operation will proceed by a sequence of shifts between layers as in the final paragraph of (1) above.

(4) *Sorting.* Sorting can proceed by the bitonic technique. The bitonic "murging" step (see Section 5.2) will require the following comparisons: $p_{k,j}$ with $p_{k,j+8}$ (once); $p_{k,j}$ with $p_{k,j+4}$ (twice); $p_{k,j}$ with $p_{k,j+2}$ (four times). All other comparisons will use either the shuffle connections within the separate layers or the connections $p_{k,j} \leftrightarrow p_{k,j+1}$, which are just as fast. The comparisons noted explicitly will require approximately 46 data moves and 100 shuffle cycles within layers, as compared to approximately 200 shuffle cycles required for sorting in an unlayered ultracomputer.

In the layered structure we have suggested, it would be reasonable to use 16 connections between processors, requiring 80K chips in a $4\frac{1}{2} \times 4 \times 2$-foot configuration. Since all connections in this structure are relatively short, it might be possible to use a communication cycle closer to 10 ns than to 20 ns; if this is so, then the time required for permutation and packing might only increase by 50 percent, the time required for summing might remain approximately constant, while the time required for sorting will be somewhat diminished.

Two-dimensional nearest neighbor communication in a simulated square array of processing elements would require something in the neighborhood of 80–120 ns.

In this configuration approximately 50 wires per processor would come off each board, for a total of approximately 5000 wires per board. Close to 32,000 wires would emerge from a cage, roughly 16,000 connecting with the preceding and 16,000 with the following cage. Moreover, the number of interconnections between cabinets would be similar.

8. CONCLUSION

In the preceding sections we have reviewed a class of architectures for very large concurrent computing assemblages, and a set of algorithms adapted to these architectures, which together seem to indicate that such assemblages can be used efficiently for a wide variety of purposes. Our discussion only indicates feasibility; better architectures and better algorithms undoubtedly exist. The programming style at which our algorithms point is one in which periods of independent parallel computation alternate with periods of interprocessor communication and data shuffling, and in which the set-theoretic primitives which we have described are used to select, group, and combine results generated by independent computation and to trigger new groups of computations which can proceed in parallel. Algorithms of this kind should be structured to maximize the amount of computation which can be done independently and minimize the need for communication; to the extent that program design attains this ideal, an N-processor ultracomputer will be able to solve a problem N times as fast as a serial processor.

This encouraging remark should apply, for example, to finite-element computations, where requirements for communication are modest and the necessary pattern of communications is known in advance. But if considerable interprocessor communication is inherently necessary to solve a problem, the relative advantage of an N-processor assemblage may fall to $N/\log N$, $N/\log^2 N$, or even lower.

Our considerations even seem to hint that ultracomputers can be controlled by

languages whose dictions are of not too unfamiliar or intimidating a character. These may, for example, involve set and map operations on ordered sets. Such operations may serve to group related set elements together within processors, after which completely independent computations may be able to proceed for a while.

If it really proves possible to define effective languages of this kind, very interesting optimization problems, whose solution might further facilitate effective use of ultracomputers, will doubtless be revealed.

Taking all in all, we come to the optimistic conclusion that effective use of extremely large parallel computing assemblages is possible. It follows that theoretical work aimed at the discovery of algorithms and languages adapted to such assemblages ought to be pushed vigorously, that LSI hardware designers ought to take cognizance of these algorithms and language efforts as their results become available and ought to try to support them, and that efforts to actually construct very large parallel computers will soon become practical.

ACKNOWLEDGMENTS

The author would like to thank Allan Gottlieb, Clyde Kruskal, L. Rudolph, and Malcolm Harrison for numerous conversations and detailed technical suggestions which have shaped the final form of this article very substantially.

REFERENCES

1. AHO, A., HOPCROFT, J., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1976.
2. BATCHER, K.E. Sorting networks and their applications. Proc. 1968 Spring JCC, AFIPS Press, Arlington, Va., pp. 307–314.
3. BAUDET, G., AND STEVENSON, D. Optimal sorting algorithms for parallel computers. Tech. Rep., Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa. See also *IEEE Trans. Comput. C-27* (1978), 84–86.
4. BENES, V.E. Mathematical theory of connecting networks and telephone traffic. Academic Press, New York, 1965.
5. CHANDRA, A.K. Maximal parallelism in matrix multiplication. Rep. RC-6193, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1976.
6. CLOS, C. A study of nonblocking switching networks. *Bell Syst. Tech. J. 32* (1953), 406–424.
7. CSANKY, L. Fast parallel matrix inversion algorithms. *SIAM J. Comput. 5* (1976), 618–623. See also Proc. 16th Ann. IEEE Symp. on Foundations of Computer Science, Berkeley, Calif., 1975, pp. 11–12.
8. GOTTLIEB, A. PLUS—A PL/I ultracomputer simulator, I. Ultracomputer Note, Computer Science Dep., New York Univ., New York, N.Y., 1980.
9. GOTTLIEB, A. PLUS—A PL/I ultracomputer simulator, II. Ultracomputer Note, Computer Science Dep., New York Univ., New York, N.Y., 1980.
10. GOTTLIEB, A., AND KRUSKAL, C.P. MULT—A multitasking ultracomputer language with timing, I. Ultracomputer Note, Computer Science Dep., New York Univ., New York, N.Y., 1980.
11. GOTTLIEB, A., AND KRUSKAL, C.P. MULT—A multitasking ultracomputer language with timing, xl 28II. Ultracomputer Note, Computer Science Dep., New York Univ., New York, N.Y., 1980.

II. Ultracomputer Note, Computer Science Dep., New York Univ., New York, N.Y., 1980.

12. GOTTLIEB, A., AND KRUSKAL, C.P. Supersaturated ultracomputer algorithms. Ultracomputer Note, Computer Science Dep., New York Univ., New York, N.Y., 1980.
13. HIRSHBERG, D.S. Parallel algorithms for the transitive closure and the connected component problems. Proc. 8th Ann. ACM Symp. on Theory of Computing, Hershey, Pa., May 1976, pp. 55–57.
14. HIRSCHBERG, D.S. Fast parallel sorting algorithms. *Commun. ACM 21*, 8 (Aug. 1978), 657–661.

See also Tech. Rep., Dep. of Electrical Engineering, Rice Univ., Houston, Texas, 1977.
15. KNUTH, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1972, pp. 232–235.
16. KUNG, H.T. New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences. *J. ACM 23*, 2 (April 1979), 252–261. See also Proc. 6th Ann. ACM Symp. on Theory of Computing, Seattle, Wash., 1974, pp. 323–333.
17. LEVITT, K.N., AND KAUTZ, W.H. Cellular arrays for the solution of graph problems. *Commun. ACM 15*, 9 (Sept. 1972), 789–801.
18. NASSIMI, D., AND SAHNI, S. Parallel permutation and sorting networks and a new generalized-connection network. Preprint, Univ. of Minnesota, Minneapolis, Minn., 1979.
19. NASSIMI, D., AND SAHNI, S. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Comput. C-28* (1979), 2–7.
20. OPFERMAN, D.C., AND TSAO-WU, N.T. On a class of rearrangeable switching networks. *Bell Syst. Tech. J. 50* (1971), 1579–1618.
21. ORCUTT, S.E. Parallel solution methods for triangular linear systems of equations. Tech. Rep. 77, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1974.
22. PEASE, M.C. Matrix inversion using parallel processing. *J. ACM 14*, 4 (Oct. 1967), 757–764.
23. PEASE, M.C. An adaptation of the fast Fourier transform for parallel processing. *J. ACM 15*, 2 (April 1968), 252–264.
24. PREPARATA, F.P. Parallelism in sorting. Proc. 1977 Int. Conf. on Parallel Processing, Detroit, Mich., 1977, pp. 202–206.
25. PREPARATA, F.P., AND SARWATE, D.V. An improved parallel processor bound in fast matrix inversion. *Inf. Proc. Letters 7* (1978), 178–150.
26. SAMEH, A., AND BRENT, R. Solving triangular systems on a parallel computer. *SIAM J. Numer. Anal.* (1977), 1101–1113.
27. STONE, H.S. Parallel processing with the perfect shuffle. *IEEE Trans. Comput. C-20* (1971), 153–161.
28. VALIANT, L.G. Parallelism in comparison problems. *SIAM J. Comput. 4* (1975), 348–355.
29. WAKSMAN, A. A permutation network. *J. ACM 15*, 1 (Jan. 1968), 159–163.

BIBLIOGRAPHY

Several useful surveys of algorithms for parallel numerical and nonnumerical computation have appeared. See Heller (1978), Miranker (1971), Kuck (1975), and Stone (1975b).

ABRAMSON, N., AND KUO, F.F., EDS. *Computer-Communication Networks.* Prentice-Hall, Englewood Cliffs, N.J., 1973.

ADAMS, D.A. A model for parallel computations. In *Parallel Processor Systems, Technologies, and Applications*, L.C. Hobbs, D.J. Theis, J. Trimble, H. Titus, and I. Highberg, Eds., Spartan Publishers, Washington, D.C., 1970, pp. 311–333.

AHO, A., AND ULLMAN, J.D. Dynamic memories with rapid random and sequential access. *IEEE Trans. Comput. C-23* (1974), 272–276.

AKERS, S.B. A rectangular logic array. *IEEE Trans. Comput. C-21* (1972), 848–857.

ANDERSON, G.A. Interconnecting a distributed processor system for avionics. Proc. IEEE Symp. on Computer Architecture, Gainesville, Fla., 1973, pp. 1–20.

ANDERSON, G.A., AND JENSEN, E.D. Computer interconnection structures: Taxonomy, characteristics, and examples. *Comput. Surv. 7*, 4 (Dec. 1975), 197–213.

ARJOMANDI, E. A study of parallelism in graph theory. Ph.D. Thesis, Dep. of Computer Science, Univ. of Toronto, Toronto, Ontario, Canada, 1977.

ARJOMANDI, E., AND CORNEIL, D.G. Parallel computations in graph theory. Proc. 16th Ann. IEEE Symp. on Foundations of Computer Science, Berkeley, Calif., 1975, pp. 13–18.

ARNOLD, R.G., AND PAGE, E.W. A hierarchical, restructurable multimicroprocessor architecture. 3rd Ann. Symp. on Computer Architecture, Clearwater, Fla., 1976, pp. 40–45.

AVRIEL, M., AND WILDE, D.J. Optimal search for a maximum with sequences of simultaneous function evaluations. *Manage. Sci. 12* (1966), 722–731.

BAER, J.L., AND RUSSELL, E.C. Preparation and evaluation of computer programs for parallel processing systems. In *Parallel Processor Systems, Technologies, and Applications*, L.C. Hobbs, D.J. Theis, J. Trimble, H. Titus, and I. Highberg, Eds., Spartan Publishers, Washington, D.C., 1970, pp. 375–415.

BANDYOPADHYAY, S., BASU, S., AND CHOUDHARY, A.K. A cellular permuter array. *IEEE Trans. Comput. C-21* (1972), 1116-1119.

BARNES, G.H., BROWN, R., KATO, M., KUCK, D., SLOTNICK, D., AND STOKES, R. The ILLIAC IV computer. *IEEE Trans. Comput. C-17* (1968), 746-757.

BASSALYGO, L.A., GRUSHKO, I.I., AND NEYMAN, V.I. Asymptotic estimation of the number of switching points in nonblocking circuits. *Telecommun. Radio Eng. 24* (1970), 34-39.

BASSALYGO, L.A., AND PINSKER, M.S. On the complexity of optimal nonblocking switching networks without rearrangement. *Probl. Inf. Transm. 9* (1973), 84-87 (translation).

BATCHER, K.E. The flip network in STARAN. Proc. 1976 Int. Conf. on Parallel Processing, Detroit, Mich., 1976, pp. 65-71.

BATCHER, K.E. STARAN series E. Proc. 1977 Int. Conf. on Parallel Processing, Wayne State Univ., Detroit, Mich., 1977, pp. 140-143.

BENES, V.E. Algebraic and topological properties of connecting networks. *Bell Syst. Tech. J. 41* (1962), 1249-1273.

BENES, V.E. Permutation groups, complexes, and rearrangeable connecting networks. *Bell Syst. Tech. J. 43* (1964), 1619-1640.

BENES, V.E. Optimal rearrangeable multistage connecting networks. *Bell Syst. Tech. J. 42* (1964), 1641-1656.

BENES, V.E. Applications of group theory to connecting networks. *Bell Syst. Tech. J. 54* (1975), 407-420.

BERNSTEIN, A. Analysis of programs for parallel processing. *IEEE Trans. Electron. Comput. EC-15* (1966), 757-763.

BOULTIS, R.L., AND FAISS, R.O. STARAN E performance and LACIE algorithms. Proc. 1977 Int. Conf. on Parallel Processing, Wayne State Univ., Detroit, Mich., 1977, pp. 144-152.

BREDT, T.H. A survey of models for parallel computing. Tech. Rep. 8, Stanford Univ. Electronics Lab., Palo Alto, Calif., 1970.

BREDT, T.H., AND MCCLUSKEY, E.J. Analysis and synthesis of control mechanisms for parallel processes. In *Parallel Processor Systems, Technologies, and Applications*, L.C. Hobbs, D.J. Theis, J. Trimble, H. Titus, and I. Highberg, Eds., Spartan Publishers, Washington, D.C., 1970, pp. 287-296.

BRENT, R. On the addition of binary numbers. *IEEE Trans. Comput. C-19* (1970), 758-759.

BRENT, R. The parallel evaluation of arithmetic expressions in logarithmic time. In *Complexity of Sequential and Parallel Numerical Algorithms*, J.F. Traub, Ed., Academic Press, New York, pp. 83-102.

BRENT, R. The parallel evaluation of general arithmetic expressions. *J. ACM 21*, 2 (April 1974), 201-206.

BRENT, R., KUCK, D., AND MARUYAMA, K. The parallel evaluation of arithmetic expressions without divisions. *IEEE Trans. Comput. C-23* (1973), 532-534.

BRINSFIELD, W.A., AND MILLER, R.E. On the composition of parallel program schemata. Conf. Rec. IEEE 12th Ann. Symp. on Switching and Automata Theory, East Lansing, Mich., 1971, pp. 20-23.

BRINSFIELD, W.A., AND MILLER, R.E. Insertion of parallel program schemata. Proc. 7th Ann. Princeton Conf. Information Sciences and Systems, Princeton, N.J., 1973.

BUDNIK, P., AND KUCK, D.J. The organization and use of parallel memories. *IEEE Trans. Comput. C-20* (1971), 1566-1569.

BUZBEE, B.L. A fast Poisson solver amenable to parallel computation. *IEEE Trans. Comput. C-22* (1973), 793-796.

CALLAHAN, D. Parallel solution of sparse simultaneous linear equations. Tech. Rep., Dep. of Electrical Engineering, Univ. of Michigan, Ann Arbor, Mich., 1973.

CANTOR, D.G. On nonblocking switching networks. *Networks 1* (1971), 367-377.

CARROLL, A.B., AND WETHERALD, R.T. Applications of parallel processing to numerical weather prediction. *J. ACM 14*, 3 (July 1967), 591-614.

CASTI, J.L., RICHARDSON, M.H., AND LARSON, R.E. Dynamic programming in parallel computers. *J. Optim. Theory Appl. 12* (1973), 423-438.

CHANDRA, A.K. Independent permutations, as related to a problem of Moser and a theorem of Polya. *J. Comb. Theory Series A 16* (1974), 111-120.

CHAZAN, D., AND MIRANKER, W.L. Chaotic relaxation. *Linear Alg. Appl. 2* (1969), 199-222.

CHAZAN, D., AND MIRANKER, W.L. A nongradient and parallel algorithm for unconstrained minimization. *SIAM J. Control 8* (1970), 207-217.

CHEN, I.N. A cellular data manipulating array. Proc. 1975 Sagamore Conf. on Parallel Processing, Sagamore Lake, N.Y., 1975, p. 114.
CHEN, S.C. Speedup of iterative programs in multiprocessor systems. Ph.D. Thesis, Computer Science Dep., Univ. of Illinois at Urbana, Urbana, Ill., 1975 [Rep. 694 (NSF-OCA-GJ-36936-000004)].
CHEN, S.C. Time and parallel processor bounds for linear recurrence systems with constant coefficients. Proc. 1976 Int. Conf. on Parallel Processing, Detroit, Mich., 1976, pp. 196–205.
CHEN, S.C., AND KUCK, D.J. Time and parallel processor bounds for linear recurrence systems. *IEEE Trans. Comput. C-24* (1975), 701–717.
CHEN, S.C., AND SAMEH, A. On parallel triangular system solvers. Proc. 1975 Sagamore Conf. on Parallel Processing, Sagamore Lake, N.Y., 1975, pp. 237–238.
CHEN, T.C. Parallelism, pipelining and computer efficiency. *Comput. Design 10* (1971), 69–74.
CHEN, T.C. Unconventional superspeed computer systems. AFIPS Proc. 1971 Spring JCC, Vol. 38, AFIPS Press, Arlington, Va., pp. 365–371.
CHEN, Y.K., AND FENG, T. A parallel algorithm for the maximum flow problem (summary). Proc. 1973 Sagamore Conf. on Parallel Processing, Sagamore Lake, N.Y., 1973, p. 60.
COLE, S.N. Real-time computation by n-dimensional arrays of finite-state machines. *IEEE Trans. Comput. C-18* (1969), 349–365.
COPPAGE, S., AND SCHWARTZ, J.T. A fast switch. *Amer. Math. Monthly 83* (1976), 711–717.
CRANE, B.A., AND GITHENS, J.A. Bulk processing in distributed logic memory. *IEEE Trans. Electron. Comput. EC-14* (1965), 186–196.
CSANKY, L. On the parallel complexity of some computational problems. Ph.D. Dissertation, Dep. of Computer Science, Univ. of California, Berkeley, Calif., 1974.
CYRE, W.R., AND LIPOVSKI, G.J. On generating multipliers for a cellular fast Fourier transform processor. *IEEE Trans. Comput. C-21* (1972), 83–87.
DAVIDSON, I.A., AND FIELD, J.A. Design criteria for a switch for a multiprocessor computing system. Proc. 1975 Sagamore Conf. on Parallel Processing, Sagamore Lake, N.Y., 1975, pp 110–113.
DENNIS, J.B., AND MISUNAS, D.P. A computer architecture for highly parallel signal processing. Proc. ACM 1974 National Conf., San Diego, Calif., pp. 402–409.
DENNIS, J.B., AND MISUNAS, D.P. A preliminary architecture for a basic data-flow processor. Proc. 2nd Ann. IEEE Symp. on Computer Architecture, Houston, Texas, 1975, pp. 126–132.
DONNELLY, J.D. Periodic chaotic relaxation. *Linear Alg. Appl. 4* (1971), 117–128.
DORN, W.S., HSU, N.C., AND RIVLIN, T.J. Some mathematical aspects of parallel computation. Tech. Rep. RC-647, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1962.
DOWNS, R.S. Real-time algorithms and data management on Illiac IV. *IEEE Trans. Comput. C-22* (1973), 773–777.
DRYSDALE, R.L. Sorting networks which generalize Batcher's odd-even merge. Honors Paper, Knox College, Galesburg, Ill., 1973.
ECKSTEIN, D. Parallel graph processing using depth-first search and breadth-first search. Ph.D. Thesis, Univ. of Iowa, 1977.
ENSLOW, P.H., ED. Multiprocessors and parallel processors. John Wiley & Sons, New York, 1974.
ERMANN, R.N., AND GROSKY, W.I. Some computation and systems-theoretic properties of regular processor networks. Proc. 1976 Int. Conf. on Parallel Processing, Detroit, Mich., 1976, pp. 230–234.
ESTRIN, G., BUSSELL, B., TURN, R., AND BIBB, J. Parallel processing in a restructurable computer system. *IEEE Trans. Electron. Comput. EC-12* (1963), 747–755.
EVEN, S. Parallelism in tape sorting. *Commun. ACM 17*, 4 (April 1974), 202–204.
EVENSEN, A.J., AND TROY, J.L. Introduction to the architecture of a 288-element PEPE. Proc. 1973 Sagamore Conf. on Parallel Processing, Sagamore Lake, N.Y., 1973, pp. 162–169.
FALKOFF, A.D. Algorithms for parallel search memories. *J. ACM 9*, 4 (Oct. 1962), 488–511.
FENG, T.Y. Data manipulating functions in parallel processors and their implementations. *IEEE Trans. Comput. C-23* (1964), 309–318.
FENG, T.Y. A configurable multi-microprocessor organization. In *Micro-Architecture of Computer Systems,* Proc. Euromicro, Nice Workshop, 1975, R. Hartenstein, Ed., North-Holland, Amsterdam, pp. 207–219.
FISHER, D. Program analysis for multi-processing. Tech. Rep. TR-67-2, Burroughs Corp., 1967.
FLOYD, R.W., AND KNUTH, D.E. The Bose–Nelson sorting problem. CS Rep. 70-177, Stanford Univ., Stanford, Calif., 1976.

FLYNN, M.J. Very-high-speed computing systems. *Proc. IEEE 54* (1966), 1901–1909.
FLYNN, M.J., PODVIN, A., AND SHIMIZU, K. A multiple instruction stream processor with shared resources. In *Parallel Processor Systems, Technologies, and Applications*, L.C. Hobbs, D.J. Theis, J. Trimble, H. Titus, and I. Highberg, Eds., Spartan Publishers, Washington, D.C., 1970, pp. 215–286.
GALE, D., AND KARP, R.K. A phenomenon in the theory of sorting. IEEE Conf. Rec. of 11th Ann. Symp. on Switching and Automata Theory, Santa Monica, Calif., 1970, pp. 51–59.
GAVRIL, F. Merging with parallel processors. *Commun. ACM 18*, 10 (Oct. 1975), 588–591.
GENTLEMAN, W.M. Some complexity results for matrix computations on parallel processors. Tech. Rep., Dep. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada, 1976.
GEORGE, A., POOLE, W.G., AND VOIGHT, R.G. Analysis of dissection algorithms for vector computers. Tech. Rep., Institute for Computer Applications in Science and Engineering, Hampton, Va., 1976.
GILL, S. Parallel programming. *Comput. J. 1* (1958), 2–10.
GILMORE, P.A. Structuring of parallel algorithms. *J. ACM 15*, 2 (April 1968), 176–192.
GILMORE, P.A. Parallel relaxation. Tech. Rep., Goodyear Aerospace Corp., Akron, Ohio, 1971.
GOKE, R. Connecting networks for partitioning polymorphic systems. Ph.D. Dissertation, Dep. of Electrical Engineering, Univ. of Florida, Gainesville, Fla., 1976.
GOKE, R., AND LIPOVSKI, G.J. Banyan networks for partitioning on multiprocessor systems. Proc. 1st Ann. Symp. on Computer Architecture, Gainesville, Fla., 1973, pp. 21–30.
GOLOMB, S.W. Permutations by cutting and shuffling. *SIAM Rev. 3* (1961), 293–297.
GONZALES, M.J., AND RAMAMOORTHY, C.V. Recognition and representation of parallel processable streams in computer programs. In *Parallel Processor Systems, Technologies and Applications*, L.C. Hobbs, D.J. Theis, J. Trimble, H. Titus, and I. Highberg, Eds., Spartan Publishers, Washington, D.C., 1970, pp. 335–373.
GONZALES, M.J., AND RAMAMOORTHY, C.V. Program suitability for parallel processing. *IEEE Trans. Comput. C-20* (1971), 647–654.
GREEN, M.W. Some improvements in nonadaptive sorting algorithms. Proc. 6th Ann. Princeton Conf. on Information Sciences and Systems, Princeton, N.J., 1972, pp. 387–391.
HARADH, K. Sequential permutation networks. *IEEE Trans. Comput. C-21* (1972), 472–479.
HELLER, D. A determinant theorem with applications to parallel algorithms. Tech. Rep., Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1973.
HELLER, D. A survey of parallel algorithms in numerical linear algebra. *SIAM Rev. 20* (1978), 740–777. See also Tech. Rep., Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1976.
HOLLAND, J. A universal computer capable of executing an arbitrary number of subprograms simultaneously. Proc. IEEE Eastern Joint Computer Conference, Boston, Mass., pp. 108–113.
HYAFIL, L., AND KUNG, H.T. Parallel algorithms for solving triangular linear systems. Tech. Rep., Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1974.
HYAFIL, L., AND KUNG, H.T. The complexity of parallel evaluation of linear recurrences. Proc. 7th Ann. ACM Symp. on Theory of Computing, Albuquerque, N.M., 1975, pp. 12–22.
HYAFIL, L., AND KUNG, H.T. Bounds on the speed-ups of parallel evaluation of recurrences. Proc. 2nd USA–Japan Computer Conf., Tokyo, 1975, pp. 178–182.
HYAFIL, L., AND KUNG, H.T. The complexity of parallel evaluation of linear recurrences. *J. ACM 24*, 3 (July 1977), 513–521.
JOEL, A.E. On permutation switching networks. *Bell Syst. Tech. J. 47* (1968), 813–822.
KARP, R.M., AND MIRANKER, W. Parallel minimax search for a maximum. *J. Comb. Theory 4* (1968), 19–35.
KARP, R.M., AND MILLER, R.E. Parallel program schemata. *J. Comput. Sys. Sci. 3* (1969), 147–195.
KARP, R.M., MILLER, R.E., AND WINOGRAD, S. The organization of computations for uniform recurrence equations. *J. ACM 14*, 3 (July 1967), 563–590.
KAUTZ, W.H. Cellular logic-in-memory arrays. *IEEE Trans. Comput. C-18* (1969), 719–727.
KAUTZ, W.H. The design of optimum interconnection networks for multiprocessors. In *Structure et Conception des Ordinateurs (Architecture and Design of Digital Computers)*, Guy Boulaye, Ed., Dunod Publishers, Paris, 1971, pp. 249–278.
KAUTZ, W.H., LEVITT, K.N., AND WAKSMAN, A. Cellular interconnection arrays. *IEEE Trans. Comput. C-17* (1968), pp. 443–451.
KAUTZ, W.H., PEASE, M.C., AND GREEN, M.W. Cellular logic-in-memory arrays. Final Rep. Pt. 1, Project 5509, NONR-4833 (00), Stanford Research Institute, Palo Alto, Calif., 1970.
KELLER, R.M. On maximally parallel schemata. Conf. Rec. 11th Ann. IEEE Symp. on Switching

and Automata Theory, Santa Monica, Calif., 1970, pp. 32–50.
KELLER, R.M. On the decomposition of asynchronous systems. Conf. Rec. 13th Ann. IEEE Symp. on Switching and Automata Theory, Baltimore, Md., 1972, pp. 78–89.
KNIGHT, J.C., POOLE, W.G., JR., AND VOIGT, R.G. System balance analysis for vector computers. Proc. ACM Ann. Conf., Minneapolis, Minn., 1975, pp. 163–168.
KOGGE, P.M. Parallel algorithms for the solution of recurrence problems. Tech. Rep., Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1972.
KOGGE, P.M. The numerical stability of parallel algorithms for solving recurrence problems. Tech. Rep., Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1972.
KOGGE, P.M. Parallel solution of recurrence problems. *IBM J. Res. Devel. 18* (1974), 183–184.
KOGGE, P.M., AND STONE, H.S. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput. C-22* (1973), 786–792.
KOTOV, V.E. Towards automatic construction of parallel programs. Proc. Symp. on Theoretical Programming, Novosibirsk, USSR, 1972, pp. 309–332.
KOTOV, V.E., AND NARINYANI, A.S. On transformation of sequential programs into asynchronous parallel programs. Proc. 1968 IFIP Congress, Edinburgh, pp. 351–357.
KUCK, D.J. Multioperation machine computational complexity. Proc. of Symp. on Complexity of Sequential and Parallel Numerical Algorithms, J.F. Traub, Ed., Academic Press, New York, pp. 17–48.
KUCK, D.J. Parallel processing architecture, a survey. Proc. 1975 Sagamore Conf. on Parallel Processing, Sagamore Lake, N.Y., 1975, pp. 15–39.
KUCK, D.J., LAURIE, D.H., AND MURAOKA, Y. Interconnection networks for processors and memories in large systems. COMPCON 72, San Francisco, Calif., 1972, pp. 131–134.
KUCK, D.J., AND MARUYAMA, K.M. The parallel evaluation of arithmetic expressions of special forms. Rep. RC4726, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1973.
KUCK, D.J., AND MARUYAMA, K.M. Time bounds on the parallel evaluation of arithmetic expressions. *SIAM J. Comput. 4* (1974), 147–162.
KUCK, D.J., AND SAMEH, A.H. Parallel computation of eigenvalues of real matrices. Proc. 1971 IFIP Congress, Vol. II, North-Holland, Amsterdam-London, pp. 1266–1272.
KUKREJA, N., AND CHEN, I.N. Combinatorial and sequential cellular structures. *IEEE Trans. Comput. C-22* (1973), 813–823.
KUNG, H.T. Synchronous and asynchronous parallel algorithms for multiprocessors. In *Algorithms and Complexity*, J.F. Traub, Ed., 1967, pp. 153–200.
LADNER, R.E., AND FISHER, M.J. Parallel prefix computations. Proc. 1977 Int. Conf. on Parallel Processing, Detroit, Mich., 1977, pp. 218–223.
LAMBIOTTE, J.J., JR., AND VOIGT, R.G. The solution of tridiagonal linear systems on the CDC STAR-100 computer. *ACM Trans. Math. Softw. 1*, 4 (Dec. 1975), 308–329.
LAMPORT, L. The parallel execution of DO loops. *Commun. ACM, 17*, 2 (Feb. 1974), 83–93.
LANG, T. Interconnections between memory modules using the shuffle-exchange network. *IEEE Trans. Comput. C-25* (1976), 496–503. See also Tech. Rep. 76, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1973.
LANG, T., AND STONE, H.S. A shuffle-exchange network with simplified control. *IEEE Trans. Comput. C-25* (1976), 55–65.
LANGE, R.G. High level language for associative and parallel computation with STARAN. Proc. 1976 Int. Conf. on Parallel Processing, Detroit, Mich., 1976, pp. 170–176.
LARSON, R.E., RICHARDSON, M.H., AND BREE, D.W. Dynamic programming in parallel computers. 4th Ann. Hawaii Conf. on Systems Science, pp. 256–269.
LARSON, R.E., RICHARDSON, M.H., AND CASTI, J.L. Dynamic programming with parallel computers for use in Air Force applications. Final Rep., AFOSR Project F44620-70-C0084, SCI Project U956, System Control, Inc., Palo Alto, Calif., 1971.
LARSON, R.E., AND TSE, E. Modal trajectory estimation and parallel computers. 2nd Symp. on Nonlinear Estimation Theory, San Diego, Calif., 1971, pp. 188–198.
LARSON, R.E., AND TSE, E. Parallel computation of the modal trajectory estimate. Joint Automata Control Conf., Stanford, Calif. See also 5th Ann. Hawaii Conf. on System Sciences, Honolulu, 1972, pp. 495–499.
LARSON, R.E., AND TSE, E. Parallel processing algorithms for the optimal control of nonlinear dynamic systems. *IEEE Trans. Comput. C-22* (1973), 777–785.

LAWRIE, D.H. Memory-processor communication networks. Tech. Rep. and Thesis, UIU CSCS-R-75-557, Computer Science Dep., University of Ill., Urbana, Ill., 1973.
LAWRIE, D.H. Access and alignment of data in an array processor. *IEEE Trans. Comput. C-24* (1975), 1145–1155.
LAWRIE, D.H., LAYMAN, T., BAER, D., AND RANDAL, J.M. GLYPNIR—A programming language for Illiac IV. *Commun. ACM 18*, 3 (March 1975), 157–164.
LEE, C.C., AND FENG, T. Sorting algorithms for parallel processing (summary). Proc. 1975 Sagamore Conf. on Parallel Processing, Sagamore Lake, N.Y., 1975, p. 239.
LEHMAN, M. A survey of problems and preliminary results concerning parallel processing and parallel processors. *Proc. IEEE 54* (1966), 1889–1901.
LEONDES, C., AND RUBINOFF, M. DINA, a digital analyzer for Laplace, Poisson, diffusion and wave equations. *AIEE Trans. (Commun. & Electron.) 71* (1952), 303–309.
LEVITT, K.N., GREEN, M.W., AND GOLDBERG, J. A study of the data communication problems in a self-repairable multiprocessor. Proc. 1968 Spring JCC, AFIPS Press, Arlington, Va., pp. 515–527.
LIPOVSKI, G.J. The architecture of a large associative processor. Proc. 1970 Spring JCC, Vol. 37, AFIPS Press, Arlington, Va., pp. 385–395.
LIPOVSKI, G.J. On a varistructured array of microprocessors. *IEEE Trans. Comput. C-26* (1977), 125–138.
LIPOVSKI, G.J., AND TRIPATHI, A. A reconfigurable varistructure array processor. Proc. 1977 Int. Conf. on Parallel Processing, Detroit, Mich., 1977, pp. 165–173.
LIU, C.L. Construction of sorting plans. In *Theory of Machines and Computation*, Z. Kohavi and A. Paz, Eds., Academic Press, New York, 1971, pp. 87–98.
LIU, J.W.H. The solution of mesh equations on a parallel computer. Tech. Rep., Dep. of Computer Science, Waterloo Univ., Waterloo, Ontario, Canada, 1974.
MADSEN, N.K., RODRIGUE, G.N., AND KARUSH, J.I. Matrix multiplication by diagonals on a vector/parallel processor. *Inf. Proc. Letters 5* (1976), 41–45.
MARCUS, M.J. New approaches to the analysis of connecting and sorting networks. Tech. Rep. 486, Research Laboratory of Electronics, M.I.T., Cambridge, Mass., 1972.
MARUYAMA, K. On the parallel evaluation of polynomials. *IEEE Trans. Comput. C-22* (1973), 2–5.
MARUYAMA, K. The parallel evaluation of matrix expressions. Tech. Rep., IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1973.
MILLER, J.C.P., AND BROWN, D.J.S. An algorithm for evaluation of remote terms in a linear recurrence relation. *Comput. J. 9* (1967), 188–190.
MILLER, R.E. A comparison of some theoretical models of parallel computation. *IEEE Trans. Comput. C-22* (1973), 710–717.
MIRANKER, W. Parallel methods for evaluating the root of a function. *IBM J. Res. Devel. 13* (1967), 297–301.
MIRANKER, W. A survey of parallelism in numerical analysis. *SIAM Rev. 13* (1971), 524–547.
MIRANKER, W., AND LINIGER, W.M. Parallel methods for the numerical integration of ordinary differential equations. *Math. Comp. 21* (1967), 303–320.
MISUNAS, D.P. A computer architecture for data-flow computation. Master's Thesis, Dep. Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., 1975.
MULLER, D.E., AND PREPARATA, F.P. Bounds to complexities of networks for sorting and for switching. *J. ACM 22*, 2 (April 1975), 195–201.
MUNRO, I., AND PATTERSON, M. Optimal algorithms for parallel polynomial evaluation. *J. Comput. Sys. Sci. 7* (1973), 189–198. See also Conf. Rec. 12th Ann. Symp. on Switching and Automata Theory, East Lansing, Mich., 1971.
MURAOKA, Y. Parallelism exposure and exploitation in programs. Ph.D. Thesis, Computer Science Dep., Univ. of Illinois, Urbana, Ill., 1971 (Tech. Rep. 424).
MURAOKA, Y., AND KUCK, D.J. On the time required for a sequence of matrix products. *Commun. ACM 16*, 1 (Jan. 1973), 22–26.
MURTHA, J.C. Highly parallel information processing systems. *Adv. Comput. 7* (1966), 2–116.
NARINYANI, A.S. Looking for a theory of parallel computation models. Proc. Symp. Theoretical Programming, Novosibirsk, USSR, 1972, pp. 247–284.
NEIMAN, V.I. Structure et commandes optimales des reseaux de connections sans blockage. *Ann. Telecommun. 24* (1969), 232–238.
NIEVERGELT, J. Parallel methods for integrating ordinary differential equations. *Commun. ACM 7*,

12 (Dec. 1964), 731–733.
OKADA, Y., TAJIMA, M., AND MORI, R. A novel multiprocessor array. 2nd Symp. on Microarchitecture (Euromicro), North-Holland, Amsterdam, 1976.
ORCUTT, S.E. Computer organization and algorithms for very high-speed computation. Ph.D. Thesis, Stanford Univ., Stanford, Calif., 1974.
PEASE, M.C. The implicit binary n-cube microprocessor array. *IEEE Trans. Comput. C-26* (1975), 153–161.
PIPPENGER, N. The complexity theory of switching networks. Tech. Rep., Research Laboratory of Electronics, M.I.T., Cambridge, Mass., 1973.
PIPPENGER, N. On the complexity of strictly nonblocking concentration networks. *IEEE Trans. Commun. COM-22* (1974), 1890–1893.
PIPPENGER, N. On crossbar switching networks. *IEEE Trans. Commun. COM-23* (1975), 646–659.
PIRTLE, M. Intercommunication of processors and memory. Proc. Fall JCC, Vol. 31, Thompson Publishing, Washington, D.C., pp. 621–633.
PREPARATA, F.P. New parallel-sorting schemes. *IEEE Trans. Comput. C-27* (1978), 669–673.
REITER, R. Scheduling parallel computations. *J. ACM 15*, 4 (Oct. 1968), 590–599.
RODRIGUEZ, J.E. A graph model for parallel computation. Ph.D. Dissertation, M.I.T., Cambridge, Mass., 1967. See also Project MAC Rep. ESL-R-398, MAC-TR-64, Sept. 1969.
RODRIGUEZ, J.E. Parallel processes, schemata, and transformations. IBM Res. Rep. RC 2912, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1970.
ROSEN, K.F. An algorithm for random walks on highly parallel machines. Tech. Rep. TR-15, Cooley Electronics Laboratory, Univ. of Michigan, Ann Arbor, Mich., 1964.
ROSENFELD, J.L. A case study in programming for parallel processors. *Commun. ACM 12*, 12 (Dec. 1969), 645–655.
ROSENFELD, J.L., AND DRISCOLL, C.G. Solution of the Dirichlet problem on a simulated parallel processing system. Proc. IFIP Congress 1968, North-Holland, Amsterdam, 1969, pp. 499–507.
SAMEH, A.H. On Jacobi and Jacobi-like algorithms for a parallel computer. *Math. Comp. 25* (1971), 579–590.
SAMEH, A.H. Linear system solvers for parallel computers. Tech. Rep., Dep. of Computer Science, Univ. of Illinois, Urbana, Ill., 1975.
SAMEH, A., AND KUCK, D. Linear system solvers for parallel computers. Rep. 701 (NSF-OCS-GJ-36936-000009), Computer Science Dep., Univ. of Illinois, Urbana, Ill., 1975.
SAMEH, A., AND KUCK, D. A parallel QR-algorithm for symmetric tridiagonal matrices. *IEEE Trans. Comput. C-26* (1977), 147–153.
SAMEH, A., AND KUCK, D. On stable parallel linear system solvers. *J. ACM 25*, 1 (Jan. 1978), 81–91.
SAVAGE, C. Parallel algorithms for graph-theoretic problems. Ph.D. Dissertation, Univ. of Illinois, Urbana, Ill., 1978.
SCHWARTZ, J. Large parallel computers. *J. ACM 13*, 1 (Jan. 1966), 25–32.
SHANNON, C.E. Memory requirements in a telephone exchange. *Bell Syst. Tech. J. 29* (1950), 347–349.
SHAPIRO, H.D. Storage schemes in parallel memories. Proc. 1975 Sagamore Conf. On Parallel Processing, Sagamore Lake, N.Y., 1975, pp. 159–164.
SHAPIRO, H.D. Theoretical limitations on the use of parallel memories. Ph.D. Thesis, Computer Science Dep., Univ. of Illinois, Urbana, Ill., 1976.
SHEDLER, G.S. Parallel numerical methods for the solution of equations. *Commun. ACM 10*, 5 (May 1967), 286–291.
SHEDLER, G.S., AND LEHMAN, M. Evaluation of redundancy in a parallel algorithm. *IBM Syst. J. 6* (1967), 142–149.
SIEGEL, H.J. Single instruction stream—multiple data stream machine interconnection network design. Proc. 1976 Int. Conf. on Parallel Processing, Detroit, Mich., 1976, pp. 272–280.
SIEGEL, H.J. Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks. *IEEE Trans. Comput. C-26* (1977), 153–161. See also Proc. 1975 Sagamore Conf. on Parallel Computing, Sagamore Lake, N.Y., 1975, pp. 106–109.
SIEGEL, H.J. The universality of various types of SIMD machine interconnection networks. Proc. IEEE 4th Ann. Symp. on Computer Architecture, Silver Springs, Md., 1977, pp. 70–79.
SIEGEL, H.J., AND SMITH, S.D. Study of multistage SIMD interconnection networks. Proc. IEEE 15th Ann. Symp. on Computer Architecture, Stanford, Calif., pp. 223–229.

SMITH, B.J. An analysis of sorting networks. Final Tech. Rep., ONR Contract N00014-70-A-0362-0006, 1972.

STONE, H.S. The organization of high-speed memory for parallel block transfer of data. *IEEE Trans. Comput. C-19* (1970), 47–53.

STONE, H.S. A logic-in-memory computer. *IEEE Trans. Comput. C-18* (1970), 719–727.

STONE, H.S. Dynamic memories with enhanced data access. *IEEE Trans. Comput. C-21* (1972), 359–366.

STONE, H.S. Problems of parallel computation. In *Complexity of Sequential and Parallel Numerical Algorithms,* J.F. Traub, Ed., Academic Press, New York, 1973, pp. 1–17.

STONE, H.S. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM 20,* 1 (Jan. 1973), 27–38.

STONE, H.S. *Discrete Mathematical Structures and Their Applications.* Science Research Associates, Chicago, Ill., 1973.

STONE, H.S. Parallel tridiagonal equation solvers. *ACM Trans. Math. Softw. 1,* 4 (Dec. 1975a), 289–307.

STONE, H.S. Parallel computers. In *Introduction to Computer Architecture,* Science Research Associates, Palo Alto, Calif., 1975b, pp. 318–374.

SULLIVAN, H., AND BASHKOW, T.R. A large scale, homogeneous, fully distributed parallel machine. Proc. 4th Annual Symp. on Computer Architecture, Silver Springs, Md., 1977, pp. 105–117.

SULLIVAN, H., BASHKOW, T.R., AND KLAPPHOLZ, D. High level language constructions in a self-organizing parallel processor. Proc. 1977 Int. Conf. on Parallel Processing, Detroit, Mich., 1977, pp. 163–174.

SULLIVAN, H., BASHKOW, T.R., AND KLAPPHOLZ, D. A large scale, homogeneous fully distributed parallel machine, II. Proc. 4th Ann. Symp. on Computer Architecture, Silver Springs, Md., 1977, pp. 118–127.

SULLIVAN, H., BASHKOW, T.R., KLAPPHOLZ, D., AND COHN, L. The node kernel: resource management in a self organizing parallel processor. Proc. 1977 Int. Conf. on Parallel Processing, Detroit, Mich., 1977, pp. 157–162.

SWANSON, R.C. Interconnection for parallel memories to unscramble *p*-ordered vectors. *IEEE Trans. Comput. C-23* (1974), 1105–1115.

THOMPSON, C.D. Generalized connection networks for parallel processor interconnection. *IEEE Trans. Comput. C-27* (1978), 1119–1125.

THOMPSON, C.D., AND KUNG, K.T. Sorting on a mesh-connected parallel computer. Proc. 8th Ann. ACM Symp. on Theory of Computing, Hershey, Pa., 1976, pp. 59–64.

THURBER, K.J. Programmable indexing networks. Proc. 1970 Spring JCC, AFIPS Press, Arlington, Va., pp. 51–58.

THURBER, K.J. Permutation switching networks. Proc. 1971 Computer Designer's Conf., Anaheim, Calif., 1971, pp. 7–24.

THURBER, K.J. Interconnection networks—A survey and assessment. AFIPS Conf. Proc., Vol. 43, AFIPS Press, Arlington, Va., pp. 909–919.

TRAUB, J.F. Iterative solution of tridiagonal systems on parallel and vector computers. In *Complexity of Sequential and Parallel Numerical Algorithms,* Academic Press, New York, 1973, pp. 49–82.

TROUT, H.R.G. Parallel techniques. Tech. Rep., Dep. of Computer Science, Univ. of Illinois, Urbana, Ill., 1972.

TSAO-WU, N.T., AND OPFERMAN, D.C. On permutation algorithms for rearrangeable switching networks. Conf. Rec. 1969 IEEE Int. Conf. on Communications, Boulder, Colo., 1969, pp. 10-29–10-34.

TSE, E. Parallel computation of the conditional mean state estimate for nonlinear systems. 2nd Symp. on Nonlinear Estimation Theory, San Diego, Calif., 1971, pp. 385–395.

TSE, E., AND LARSON, R.E. Optimum quantization and parallel algorithms for nonlinear state estimation. 3rd Symp. on Nonlinear Estimation Theory, San Diego, Calif., 1972.

TUTTLE, P.G. Implementation of selected eigenvalue algorithms on a vector computer. Master's Thesis, Univ. of Virginia, Charlottesville, Va., 1975.

VALIANT, L.G. The intrinsic complexity of parallelism in comparison problems. Tech. Rep., Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1974.

VAN VOORHIS, D.C. A lower bound for sorting networks that use the divide-sort-merge strategy. Tech. Rep. 17, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1971.

VAN VOORHIS, D.C. A generalization of the divide-sort-merge strategy for sorting networks. Tech. Rep. 16, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1971.
VAN VOORHIS, D.C. Large [g, d] sorting networks. Tech. Rep. 18, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1971.
VAN VOORHIS, D.C. Toward a lower bound for sorting networks. In *Complexity of Computer Computations*, Plenum Publishing, New York, 1972, pp. 119–129.
VAN VOORHIS, D.C. An economical construction for sorting networks. Working Paper 16/A45 No. 1, IBM Systems Development Division, Los Gatos, Calif. See also Proc. 1974 NCC, AFIPS Press, Arlington, Va., pp. 921–926.
WAKSMAN, A. On permutation networks. Proc. Hawaii Int. Conf. on System Sciences, Univ. of Hawaii, Honolulu, 1968, pp. 581–582.
WARD, R.C. The QR algorithm and Hyman's method on vector computers. *Math. Comput. 30* (1976), 132–142.
WEN, K.Y. Interprocessor connection—Capabilities, exploitation and effectiveness. Ph.D. Thesis, 1976.
WINOGRAD, S. On the time required to perform addition. *J. ACM 12*, 2 (April 1965), 277–285.
WINOGRAD, S. On the time required to perform multiplication. *J. ACM 14*, 4 (Oct. 1967), 793–802.
WITTIE, L.D. Efficient message routing in mega-microcomputer networks. Proc. IEEE 3rd Ann. Symp. on Computer Architecture, Clearwater, Fla., 1976, pp. 136–140.
WONG, C.K., AND YUE, P.C. Anticipatory control on a permutable memory. *IEEE Trans. Comput. C-22* (1973), 481–488.
WULF, W., AND BELL, C.G. C.mmp—A multi-mini-processor. Proc. AFIPS 1972 Fall JCC, Vol. 41, AFIPS Press, Arlington, Va., pp. 765–778.

Received February 1979; revised March and April 1980; accepted May 1980