PB-234 513

A NOVEL PARALLEL COMPUTER ARCHITECTURE AND SOME APPLICATIONS

Samuel E. Orcutt

Stanford University

Prepared for:

National Science Foundation

May 1974

i

BIBLIOGRAPHIC DATA SHEET	1. Report No. Technical Report No. 71	2.	PB 234 513

4. Title and Subtitle A Novel Parallel Computer Architecture and Some Applications	5. Report Date May 1974 6.
7. Author(s) Samuel E. Orcutt	8. Performing Organization Rept. No.
9. Performing Organization Name and Address Stanford University Digital Systems Laboratory Stanford, California 94305	10. Project/Task/Work Unit No. 11. Contract/Grant No. NSF Grant GJ-41093
12. Sponsoring Organization Name and Address National Science Foundation 1800 G Street NW Washington, D.C. 20550	13. Type of Report & Period Covered technical 14.

15. Supplementary Notes

STAN-CS-74-430

16. Abstracts

(see attached sheets)

17. Key Words and Document Analysis. 17a. Descriptors

parallel computation, computer architecture, interconnection networks, triangular linear systems, matrix multiplication, sorting.

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Approved for public release; distribution unlimited	UNCLASSIFIED	47
	20. Security Class (This Page) UNCLASSIFIED	22. Price $5.50 / 1.45

ABSTRACT

Present day computers have been designed with processing elements and memories in a one-to-one correspondence. For many problems this architecture limits the speed of solution. In this paper a machine architecture is presented in which processing elements and memories are considered independent resources. This architecture provides a technique for increasing the logical bandwidth of the memory without increasing the physical bandwidth. The scheme for interconnecting processing elements and memories is based on the mathematical formulation of a matrix-matrix product.

Of interest in determining the usefulness of a particular computer architecture are the problem classes which it is able to solve efficiently. For this machine we consider several problems. On a serial processor the multiplication of two $n \times n$ matrices requires $O(n^3)$ steps when using the classical algorithm or $O(n^{(\log_2 7)})$ steps when using Strassen's algorithm. We present an algorithm for our machine which performs this multiplication in $O(\log n)$ steps. This can easily be shown to be the minimum time possible. We also consider the solution of linear triangular systems of equations. This problem requires $O(n^2)$ steps for a serial processor, and $O(n)$ steps for a parallel processor of the ILLIAC IV-type. We present a parallel algorithm suited to execution on our machine which solves these systems in $O(\log^2 n)$. This algorithm is based on an extension of the principle of recursive doubling.

STAN-CS-74-430

iii

A Novel Parallel Computer Architecture and Some Applications

by

Samuel E. Orcutt

May 1974

Technical Report No. 71

Digital Systems Laboratory

Stanford Electronics Laboratories

Stanford, California

This work was supported by Bell Laboratories and by the National Science Foundation under grant GJ-41093.

A NOVEL PARALLEL COMPUTER ARCHITECTURE AND SOME APPLICATIONS

by Samuel E. Orcutt

Abstract

Present day computers have been designed with processing elements and memories in a one-to-one correspondence. For many problems this architecture limits the speed of solution. In this paper a machine architecture is presented in which processing elements and memories are considered independent resources. This architecture provides a technique for increasing the logical bandwidth of the memory without increasing the physical bandwidth. The scheme for interconnecting processing elements and memories is based on the mathematical formulation of a matrix-matrix product.

Of interest in determining the usefulness of a particular computer architecture are the problem classes which it is able to solve efficiently. For this machine we consider several problems. On a serial processor the multiplication of two $n \times n$ matrices requires $O(n^3)$ steps when using the classical algorithm or $O(n^{(\log_2 7)})$ steps when using Strassen's algorithm. We present an algorithm for our machine which performs this multiplication in $O(\log n)$ steps. This can easily be shown to be the minimum time possible. We also consider the solution of linear triangular systems of equations. This problem requires $O(n^2)$ steps for a serial processor, and $O(n)$ steps for a parallel processor of the ILLIAC IV-type.

We present a parallel algorithm suited to execution on our machine which solves these systems in $O(\log^2 n)$. This algorithm is based on an extension of the principle of recursive doubling.

In addition to numerical type algorithms, we present algorithms for several combinatorial type problems. In particular, we give methods for performing permutations and sorting. These algorithms require $O(\log n)$ steps when operating on n items.

I. INTRODUCTION

Present day computers have been designed with processing elements (PE's) and memories in a one-to-one correspondence. The classical serial processor is composed of a single PE and a single memory. In a typical parallel processor, say the ILLIAC IV, there are a multplicity of individual processors, each one composed of a PE and a memory. In this case there are many independent PE's and memories but they are organized so as to associate a single memory with each PE.

As an alternative, PE's and memories can be interconnected in an arbitrary manner. In Section II a machine architecture that makes use of this additional freedom is described. This architecture provides a method for increasing the logical bandwidth of the memory without increasing the physical bandwidth. The scheme used for interconnecting PE's and memories is based upon the mathematical formulation of matrix-matrix products.

In order to utilize such a computer organization effectively, it is essential that the algorithms chosen be organized in a manner appropriate to this organization. Considerable recent research has been done on the development of algorithms in this manner, much of the impetus in this area being promoted by the ILLIAC IV project. The main emphasis of this research has been toward developing algorithms for solving problems of size n, size being measured in a manner appropriate to the particular problem under consideration, on a machine with n processors.

In Section III we present an algorithm for computation of matrix-matrix products that requires only $O(\log n)$ steps when executed on our machine. In

Section IV the speedup and efficiency of this algorithm is investigated. We also consider the problem of solving triangular linear systems of equations. The serial algorithm requires $O(n^2)$ steps, and the straightforward parallel algorithm requires $O(n)$ steps when executed on a parallel machine of the ILLIAC IV-type. The fastest algorithm known to this author is that of Heller [1973]. By applying matrix theoretical arguments to a lower Hessenberg matrix derived from the original triangular matrix, he develops an algorithm which requires $O(\log^2 n)$ when executed on a MIMD computer with $O(n^4)$ processors and an appropriate interconnection network. (We use MIMD and SIMD in the sense of Flynn [1966]. They are taken to mean Multiple Instruction stream - Multiple Data stream and Single Instruction stream - Multiple Data stream respectively.) By applying recursive doubling arguments, similar to those of Stone [1973a] and Kogge [1974], directly to the recurrence relation represented by the triangular system we also develop an algorithm requiring $O(\log^2 n)$ steps. Although this time behavior is identical to Heller's, our algorithm requires only $O(n^3)$ processors and is suitable for execution on the machine of Section II.

In Section V we give serial and straightforward parallel algorithms for the solution to triangular linear systems. Section VI presents the basic principle upon which our algorithm is based, the principle of recursive doubling developed by Stone. In Sections VII and VIII we develop our algorithm for the solution to triangular linear systems of equations. This problems is of interest as it forms a fundamental step in the solution of general linear systems of equations when using the LU factorization.

In addition to the numerical type problems, it is useful to consider some combinatorial applications. The two problems considered in Section IX are sorting and permutations. For the case of sorting the best known parallel algorithm takes $O(\log^2 n)$ steps, while the algorithm we present requires only $O(\log n)$ steps. The algorithm presented for performing permutations also requires $O(\log n)$ steps.

II. THE MACHINE ARCHITECTURE

Extant parallel computers have been designed using a set of independent processors, each consisting of a PE connected to a memory. Although this architecture is suitable for many applications, there exist large problem classes that are not well suited to efficient solution on such a machine. To solve problems of these classes efficiently, a computer architecture more versatile than those currently in use must be developed. In this section such an architecture, based upon a set of independent PE's, a set of independent memories, and a set of interconnections, is considered. A block diagram of this architecture is shown in Figure II.1. The five blocks in the diagram are each described in detail in the remainder of this section.

A bottleneck in many computers is the memory. To compute at maximum possible speed the memory must have sufficient bandwidth to supply the arithmetic hardware with operands as fast as they are used. In modern computers several techniques have been used to increase the availability memory bandwidth without just buying faster and faster memory. The most commonly used technique is memory interleaving. This consists of partitioning a single memory into a set of smaller, independently operating modules. In this way multiple memory requests may be simultaneously active resulting in a higher bandwidth than otherwise. We have taken this principle one step further. In addition to having multiple independent memory modules, we arrange the data paths from the memories in such a way as to transmit the data from a single memory to many different PE's simultaneously.

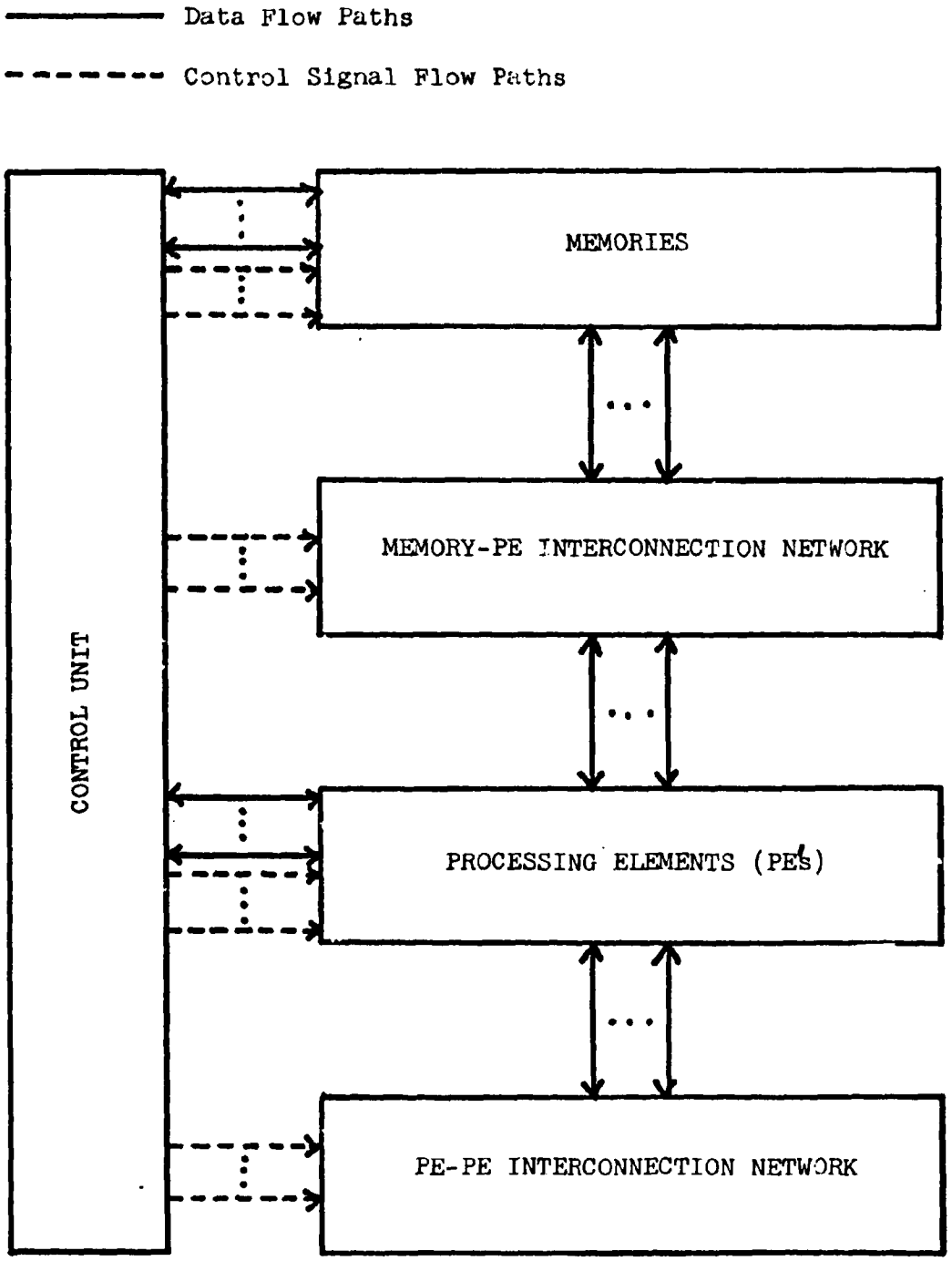

Figure II.1

Basic Machine Block Diagram

In this way, if n PE's are connected to each memory we obtain an increase in memory bandwidth of up to n.

There are n^2 memories in our computer. Conceptually these memories are organized as a n×n array. Each memory is assigned a unique label (i,j) where $0 \leq i,j \leq n-1$. Every memory can be addressed independently of all other memories. The memories operate synchronously, and any or all of the memories may be active during any memory cycle.

Memory addressing on this machine is similar to that on conventional machines with interleaved memories. To identify a single word in memory it is necessary to specify both the memory in which the word is located and its displacement within that memory. The way in which a memory address is formed from these two items is shown schematically in Figure II.2. For our machine there are two different cases. When a memory address is obtained from the PE array the memory identification is implicitly provided from the label of the PE providing the address in a manner described later in this section. In this case only the displacement need be specified. When access is from the control unit both the memory identification and displacement must be explicitly provided.

There are n^3 PE's in our computer. Conceptually these PE's are organized as a $n \times n \times n$ array. Each PE is assigned a unique label (i,j,k) where $0 \leq i,j,k \leq n-1$. The PE's each have the basic arithmetic capabilities normally found in a serial machine. All PE's obtain their instructions simultaneously from a single instruction stream. A PE is either enabled or disabled from executing instructions according to its local enable bit. These enable bits are set and reset by either local tests or global enable-setting instructions.

Figure II.2

Structure of Memory Addresses

The control unit (CU) is the central element in the computer structure. It is a computer in its own right. The instruction repertoire of the CU consists of most instructions found in typical serial machine repertoires, plus instructions for controlling the parallel features of our machine. The instruction stream of the machine is under control of the CU. Instructions that pertain to the CU are executed locally. Instructions that pertain to other parts of the machine are decoded in the CU and sent to the appropriate part of the machine in the form of control signals. Included among these control signals are any common operands required by the PE's.

The memory - PE interconnection network provides the requisite data transfer paths between the memories and the PE's. Which PE's are connected to each memory depends on which of three memory access modes is used. In any case, n PE's are simultaneously connected to every memory. Table II.1 describes the connections present for memory (i,j) in each of the access modes. In this table, an entry of the form (i,j,*) represents the set of all PE's whose labels have i and j as the first and second components respectively.

The PE-PE interconnection network provides the requisite processor interconnections. As in the case of the memory-PE interconnection network, there are three modes of operation. In each of these modes the PE's are partitioned into n <u>blocks</u> of n^2 PE's each. Data is exchanged only between PE's in the same block. These exchanges take place simultaneously and in an identical manner for all n blocks. Figure II.3 shows the interconnections established within a typical block of PE's when the interconnection network is used in mode Z. The interconnections are similar in the other two modes.

MODE	PE Providing Addresses	PEs Receiving Data When Fetching	PE Providing Data When Storing
X	(0,i,j)	(*,i,j)	(0,i,j)
Y	(i,0,j)	(i,*,j)	(i,0,j)
Z	(i,j,0)	(i,j,*)	(i,j,0)

Table II.1

Memory-PE Interconnection Network Function

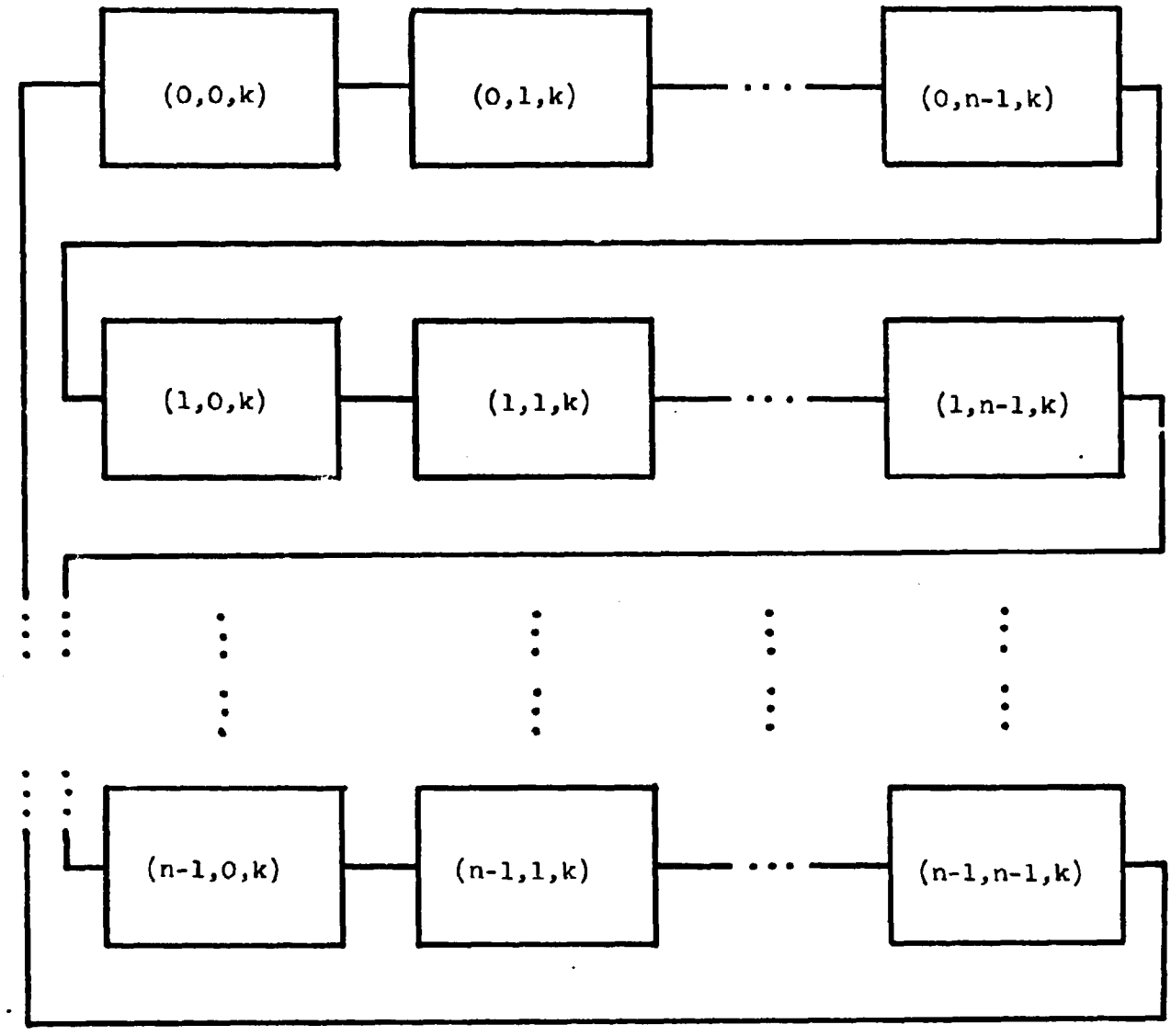

Figure II.3

Processor Interconnections for a typical block when in mode = Z

Table II.2 describes these interconnections more precisely. This table specifies the PE (i',j',k') that is at a distance d from PE (i,j,k) in each of the three modes, that is if data is routed a distance d, the datum from PE (i,j,k) ends up in PE (i',j',k'). In this table, an entry of the form (i,*,*) represents the set of all PE's with labels whose first component is i.

To further clarify the nature of the machine we now give examples of several typical machine instructions. A more complete description of the machine instructions is given in Appendix 1. The instructions are described in a notation very similar to the ISP notation of Bell and Newell [1971]. The main deviation from ISP is an extension to allow description of the explicit parallel activities that occur in the machine. To this end we define a new type of item that we call an index variable. An index variable has as its value a set of integers. The appearance of an index variable in a statement indicates that the statement is to be executed simultaneously for every integer in the set associated with that index variable. Index variables do not correspond to anything physically present in the machine architecture but are merely a notational convenience. In these descriptions we use the following symbolic coding conventions. $PERi is the symbolic name of Processing Element Register i, $CURi is the symbolic name of Control Unit Register i, and $R is the symbolic name of the Route register. The mnemonics used here are not necessarily those that should be present in an actual assembly language but are chosen for their descriptive quality. Since some of the PE and CU instructions have identical mnemonics, we assume that the actual bit encoding of the instructions allows the type of instruction, PE or CU, to be determined. We present CU instructions first.

MODE	i'	j'	k'	BLOCKS
X	i	((n·j+k+d)/n) mod n	(k+d) mod n	(i,*,*)
Y	((n·i+k+d)/n) mod n	j	(k+d) mod n	(*,j,*)
Z	((n·i+j+d)/n) mod n	(j+d) mod n	k	(*,*,k)

Table II.2

PE (i',j',k') that is at distance d from PE (i,j,k) in each mode.

Example Instruction		Comment
LOAD	$CUR1,N	$CUR1 := MEMORY(N)
SUB	$CUR0,$CUR2	$CUR0 := $CUR0 - $CUR2
JMP	EQ,LAB1	if condition code indicates equal then go to LAB1
BCAST	$CUR2,$PER1	for every PE that is enabled do $PER1 := $CUR2

The instructions LOAD, SUB, and JMP function like similar instructions in a serial machine. The instruction BCAST is used to broadcast a single datum to all the PE's.

We now consider PE instructions. All of these instructions, except ROUTE, affect only PE's that are enabled. ROUTE affects all PE's.

Example Instruction		Comment
LOAD,X	$PER1,A	$PER1 := MEMORY(A) in access mode X
ADD	$PER0,$PER3	$PER0 := $PER0 + $PER3
ROUTE,Y	3	route data between PE's a distance d = 3 in routing mode Y
CMP	$PER1,$PER2	the condition code is set according to $PER1 : $PER2
SETE	A,GE	set the enable bit if and only if the condition code indicates > or =.

These instructions are basically similar to typical parallel computer instructions. The differences arise from the strategy used for memory accessing and PE interconnections. The exact nature of these two items was described previously in this section.

III. COMPUTATION OF MATRIX - MATRIX PRODUCTS

For an algorithm to execute efficiently on a given computer the organization of the algorithm must be suited to the structure of the computer. To utilize the available computing resources as efficiently as possible, algorithms should be selected with this in mind. As an example, consider the computation of a matrix-matrix product.

Let A, B, and C be $n \times n$ matrices. Consider the equation

$$C = A \cdot B$$

This computation is to be performed in minimal time using arithmetic operations on pairs of operands. It is easily shown, by a fan-in argument, that the minimal time required for this computation is $O(\log n)$. By introducing the maximum apparent parallelism into the classical serial algorithm for computing matrix - matrix products, a parallel algorithm that requires $O(\log n)$ steps when executed on the machine of Section II is developed.

It is well known that

$$c_{ik} = \sum_{j=0}^{n-1} a_{ij} \cdot b_{jk} \qquad 0 \le i,k \le n-1$$

The computation of $a_{ij} \cdot b_{jk}$ is performed in PE (i,j,k). In this way all n^3 products can be computed simultaneously provided sufficient operands are available. To obtain these operands the matrices A, B, and C must be stored in a manner that allows access to the entire matrix simultaneously, and that allows each PE access to the proper elements of the matrices. It

is easily verified that both these criteria are satisfied if the $(i,j)^{th}$ elements of A, B, and C are all stored in memory (i,j).

To complete the computation of the matrix - matrix product, the summations

$$\sum_{j=0}^{n-1} (a_{ij} \cdot b_{jk}) \qquad 0 \le i,k \le n-1$$

must be evaluated. The interconnections provided are sufficient to allow the computation of these summations. The following algorithm, where ACC[i,j,k] is an accumulator, one of the $PER's, of PE (i,j,k), computes a matrix - matrix product. The algorithm is given in an ALGOL-like notation [Stone, 1973a]. In this notation an inequality of the form $(r \le i \le s)$ following a statement means the statement is to be executed simultaneously for all values of the index in the specified range.

```
array A,B,C[0: n-1,0: n-1];
ACC[i,j,k] := A[i,j] × B[j,k], (0 ≤ i ≤ n-1), (0 ≤ i ≤ n-1),
                    (0 ≤ k ≤ n-1);
for ii := 1 step ii until n/2 do
     ACC[i,j,k] := ACC[i,j,k] + ACC[i,j+ii,k], (0 ≤ i ≤ n-1),
                         (0 ≤ j ≤ n-1),  (0 ≤ k ≤ n-1);
C[i,k] := ACC[i,0,k],  (0 ≤ i ≤ n-1),  (0 ≤ k ≤ n-1);
```

The function of this algorithm is quite simple to understand. We first form all n^3 products $a_{ij} \cdot b_{jk}$. We then apply the log-sum algorithm n^2 times in parallel to compute each of the c_{ik}.

To help illustrate the use of our machine, we now give a machine language equivalent of this algorithm.

```
        SETE                            %Enable all PEs
        LOAD,Z      $PER1,A             %PE(i,j,k) gets
        LOAD,X      $PER2,B             %PE(i,j,k) gets B(j,k)
        MPY         $PER1,$PER2         %form A(i,j) * B(j,k)
        LOAD        $CUR1, '1'          %Set ii=1
        LOAD        $CUR2,ndiv2         %ndiv2 = n/2
        JMP         ,*+2                %Skip increment first time through
L1:     ADD         $CUR1,$CUR1         %ii := ii + ii
        CMP         $CUR1,$CUR2
        JMP         GT,L2               %go to L2 if ii > n/2
        LOADR       $R,$PER1            %load register to prepare for route
        ROUTE,Z     $CUR1
        ADD         $PER1,$R            %step of log sum
        JMP         ,L1
L2:     STORE,Y     $PER1,C             %store results
```

IV. SPEEDUP AND EFFICIENCY

One of the major reasons for building a parallel computer is to increase the throughout, measured in an appropriate manner, of the computer as compared with that obtainable with a serial machine using equivalent technology. Although the maximum obtainable speedup is determined by the architecture of the machine, the actual speedup obtained is determined by the particular algorithm being considered.

Consider the parallel computer described in Section II. With n^3 PE's the maximum obtainable speedup is n^3. For the classical matrix - matrix product algorithm $O(n^3)$ steps are required in the serial case while the parallel version requires only $O(\log n)$ steps, assuming that the time required for a maximal length route is comparable to the time required for a typical arithmetic operation (For the ILLIAC IV a maximal length route takes 14 clocks while a typical floating multiply takes 9 clocks.) In this case the interconnections provided are only slightly sub-optimal and consequently matrix - matrix products can be computed in nearly minimal time. This gives a speedup of $O(n^3/\log n)$.

Although this is the speedup obtained for the classical algorithm, it is not the speedup obtained for matrix - matrix products in general. Recent work by Winograd [1968] and Strassen [1969] has shown that matrix - matrix products can be evaluated in less than $O(n^3)$ steps. Strassen presents an algorithm requiring only $O(n^{(\log_2 7)})$ steps. This reduces the speedup obtained for matrix - matrix products to $O(n^{(\log_2 7)}/\log n)$.

Although speedup is important, it is not as important as cost-effectiveness when evaluating a parallel computer. One of the main factors entering into the cost-effectiveness is resource utilization. For the parallel matrix-matrix product this is easily determined. In this analysis we consider relative utilization as compared with that for a serial machine. Resource utilization is evaluated as (results generated / unit time) / unit of hardware. We consider two cases. For the classical serial matrix-matrix product we obtain

$$\text{PE} = (n^2 \,/\, O(n^3)) \,/\, 1 = O(1/n)$$

$$\text{Memory} = (n^2 \,/\, O(n^3)) \,/\, 1 = O(1/n)$$

For Strassen's method we obtain

$$\text{PE} = (n^2 \,/\, O(n^{(\log_2 7)})) \,/\, 1 \approx O(1/n^{0.8})$$

$$\text{Memory} = (n^2 \,/\, O(n^{(\log_2 7)})) \,/\, 1 \approx O(1/n^{0.8})$$

For the parallel algorithm the results are

$$\text{PE} = (n^2 \,/\, O(\log_2 n)) \,/\, n^3 = O(1/(n \cdot \log n))$$

$$\text{Memory} = (n^2 \,/\, O(\log_2 n)) \,/\, n^2 = O(1/\log n)$$

From these figures we can determine the relative efficiency of resource utilization for the parallel computer. The figures that we give are obtained by comparing with the Strassen algorithm. If the comparison were made with the classical algorithm the figures would be approximately $n^{0.2}$ higher.

$$\text{PE} \approx O(1/(n \cdot \log n))/O(1/n^{0.8}) = O(1/(n^{0.2} \cdot \log n))$$

$$\text{Memory} \approx O(1/\log n)/O(1/n^{0.8}) = O(n^{0.8}/\log n)$$

With present technology the main cost of the parallel computer would probably be involved in the memory. Since the memory utilization of the machine is quite good, the cost-effectiveness of this architecture should be reasonably high. As for the PE's, although their utilization is low current trends in integrated circuit technology indicate that the costs of the PE's should be quite low, allowing them to be used rather inefficiently without causing the overall cost-effectiveness to to be lowered significantly.

V. TRIANGULAR LINEAR SYSTEMS OF EQUATIONS

For a computer to be useful it should be capable of solving more than a single class of problems. We now consider the problem of solving triangular linear systems of equations on the machine of Section II.

We wish to solve problems of the form

$$M\ y = b$$

where

$$M = \begin{bmatrix} m_{11} & & & & \\ m_{21} & m_{22} & & & \\ m_{31} & m_{32} & m_{33} & & \\ \cdots & \cdots & \cdots & \cdots & \\ m_{n1} & m_{n2} & m_{n3} & \cdots & m_{nn} \end{bmatrix}$$

For the purpose of our derivation we choose to work with an equivalent formulation. Consider evaluating sequences of the form

$$y(i) = \sum_{j=0}^{i-1} A(i,j) \cdot y(i) + H(i) \qquad 0 \le i \le N$$

A, H, and N are related to M, b, and n by

$$H(i) = b_{i+1} \ / \ m_{i+1,i+1}$$

$$A(i,j) = -m_{i+1,j+1} \ / \ m_{i+1,i+1} \qquad 0 \le j \le i-1$$

$$N = n - 1$$

These sequences can be easily evaluated on a serial computer in the following manner.

```
for i := 0 step 1 until N do y(i) := H(i);
for j := 0 step 1 until N-1 do
    for i := j+1 step 1 until N do
        y(i) := y(i) + A(i,j) × y(j);
```

This algorithm requires $N\cdot(N+1)/2$ each of additions and multiplications. The algorithm thus requires $O(N^2)$ steps.

From the above serial algorithm we derive the following parallel algorithm for a machine of the ILLIAC IV-type.

```
y(i) := H(i),    (0 ≤ i ≤ N);
for j := 0 step 1 until N-1 do
    y(i) := y(i) + A(i,j) × y(j),    (j+1 ≤ i ≤ N);
```

This algorithm requires N each of addition and multiplication steps, each consisting of up to N operations performed simultaneously in parallel. This yields a speedup of $(N+1)/2$ as compared with the sequential algorithm.

We would like to obtain further speedup of the algorithm but there appears to be no straightforward way in which this speedup can be obtained. For each value of j the statement in the for loop requires the values of y from the previous iteration. This situation is quite similar to that encountered in Stone [1973a] for tridiagonal systems. On this basis we apply the techniques of recursive doubling to our problem.

VI. THE BASIC PRINCIPLE

The basic principle used in the development of our algorithm is an extension of the technique termed recursive doubling by Stone [1973a]. This technique is discussed in detail in Kogge [1974], and the interested reader is referred there for a more thorough discussion. By way of an example, we now present sufficient background to enable the reader to understand the derivation of the next section.

Consider the problem of evaluating $y(i)$, $0 \le i \le N$, where $y(i)$ is defined by the linear recurrence

$$y(0) = y_0$$

$$y(i) = A(i) * y(i-1) \quad , \quad i \ge 1$$

We proceed by deriving a sequence of equations for $y(i)$ of the following form.

$$y(i) = A^{(k)}(i) * y(i-2^k) \quad , \quad A^{(k)}(i) = A^{(k-1)}(i) * A^{(k-1)}(i-2^{k-1})$$

These equations, although valid in general, must be modified slightly to account for the boundary conditions in the recurrence that occur at $y(0)$. In this way we derive a method for computing the values of $y(i)$, $1 \le i \le N$. Let $n = \lceil \log_2 N \rceil$. We evaluate $A^{(n)}(i)$, $1 \le i \le N$, according to the above formular. From the definition of $A^{(n)}(i)$ we know that

$$y(i) = A^{(n)}(i) * y_0 \qquad 1 \le i \le N$$

Thus, after computing the first $\lceil \log_2 N \rceil$ sets of $A^{(k)}(i)$, the values of $y(i)$, $1 \le i \le N$, are all available as the result of a single multiplication. On an SIMD computer with appropriate interconnections this can be done in $O(\log N)$ time and requires $O(N)$ processors. We now proceed with our main development.

VII. A DOUBLING FORMULA

Consider evaluating $y(i)$, $0 \le i \le N$, where $y(i)$ is defined by the inhomogeneous linear recurrence

$$y(i) = \sum_{j=0}^{i-1} A(i,j) * y(j) + H(i) \qquad i \ge 0$$

We proceed, in the same manner as for our previous example, to derive a sequence of equations for $y(i)$ of the form

$$(*) \qquad y(i) = \sum_{j=0}^{i-2^k} A^{(k)}(i,j) * y(j) + H^{(k)}(i) \qquad i \ge 0$$

A vacuous sum is interpreted as having the constant value 0. From these equations we can evaluate $y(i)$ by

$$y(i) = H^{(k)}(i) \ , \ k \ge \lceil \log_2 (i+1) \rceil$$

To evaluate $y(i)$, $0 \le i \le N$, we derive the $\lceil \log_2 (N+1) \rceil^{st}$ set of equations for $y(i)$. We now give an inductive proof of the validity of (*) which yields appropriate recurrence relations defining $A^{(k)}(i,j)$ and $H^{(k)}(i)$.

Basis Step:

Let $A^{(0)}(i,j) = A(i,j)$ and $H^{(0)}(i) = H(i)$

From this we have immediately

$$y(i) = \sum_{j=0}^{i-2^0} A^{(0)}(i,j) * y(j) + H^{(0)}(i) \ , \ i \ge 0$$

Induction Step:

Assume that (*) is valid for k = n-1. We prove that (*) is also valid for k = n. We know that

$$y(i) = \sum_{j=0}^{i-2^{n-1}} A^{(n-1)}(i,j) * y(j) + H^{(n-1)}(i)$$

Using the inductive hypothesis we substitute for $y(i-2^{n-1})$, $y(i-2^{n-1}-1)$, ..., $y(i-2^{n}+1)$. This yields the following recurrence relations defining $A^{(n)}(i,j)$ and $H^{(n)}(i)$.

Case 1: $i \geq 2^{n}+2^{n-1}-1$

$$H^{(n)}(i) = H^{(n-1)}(i) + \sum_{j=i-2^{n}+1}^{i-2^{n-1}} A^{(n-1)}(i,j) * H^{(n-1)}(j)$$

$$A^{(n)}(i,j) = \begin{cases} A^{(n-1)}(i,j) + \sum_{k=i-2^{n}+1}^{i-2^{n-1}} A^{(n-1)}(i,k) * A^{(n-1)}(k,j) \\ \qquad 0 \leq j \leq i-2^{n}+1-2^{n-1} \\ A^{(n-1)}(i,j) + \sum_{k=j+2^{n-1}}^{i-2^{n-1}} A^{(n-1)}(i,k) * A^{(n-1)}(k,j) \\ \qquad 1-2^{n}+2-2^{n-1} \leq j \leq i-2^{n} \end{cases}$$

Case 2: $2^n \le i < 2^n+2^{n-1}-1$

$$H^{(n)}_{(i)} = H^{(n-1)}_{(i)} + \sum_{j=i-2^n+1}^{i-2^{n-1}} A^{(n-1)}_{(i,j)} * H^{(n-1)}_{(j)}$$

$$A^{(n)}_{(i,j)} = A^{(n-1)}_{(i,j)} + \sum_{k=j+2^{n-1}}^{i-2^{n-1}} A^{(n-1)}_{(i,k)} * A^{(n-1)}_{(k,j)}$$

$$0 \le j \le i-2^n$$

Case 3: $2^{n-1} \le i < 2^n$

$$H^{(n)}_{(i)} = H^{(n-1)}_{(i)} + \sum_{j=0}^{i-2^{n-1}} A^{(n-1)}_{(i,j)} * H^{(n-1)}_{(j)}$$

Case 4: $i < 2^{n-1}$

$$H^{(n)}_{(i)} = H^{(n-1)}_{(i)}$$

The four cases shown above can be reduced by noting that the differences between the cases are due to different bounds on otherwise identical summations.

VIII. THE ALGORITHM

We now present the algorithm used to evaluate the y(i). This algorithm is obtained directly from the recurrence relations for $A^{(k)}(i,j)$ and $H^{(k)}(i)$ derived in the previous section. In this algorithm the arrays A and H contain the current values of $A^{(k)}(i,j)$ and $H^{(k)}(i)$, and the procedure P computes vector inner products. The language that we use here is slightly different from ALGOL in the declaration and usage of arrays as parameters of procedures. A declaration of a formal parameter as array A[*] indicates that the corresponding actual parameter is to be a one-dimensional array. Similarly, actual parameters of the form A[i,*] and A[*,j] denote the i^{th} row and j^{th} column of an array A.

```
1    begin
2    real array A[0:N,0:N];
3    real array H[0:N];
4    real procedure P(U,V,FIRST,LAST);
5        real array U,V[*];
6        integer FIRST,LAST;
7        begin
8        real array T[0:N]; comment T is used for temporary storage;
9        T[k] := 0., (0 ≤ k ≤ N);
10       T[k] := U[k] × V[k], (FIRST ≤ k ≤ LAST);
11       for ℓ:= 1 step ℓ until N do
```

```
12              T[k] := T[k] + T[k+ℓ],  (0 ≤ k ≤ N-ℓ);
13          P := T[0];
14          end of procedure P;
15      INITIALIZE; comment this procedure initializes A and H;
16      for m := 1 step m until N do
17          begin
18          H[i] := H[i] + P(A[i,*], H[*],max(0,i-2 × m+1),i-m),
19              (m ≤ i ≤ N);
20          A[i,j] := A[i,j] + P(A[i,*],A[*,j],max(i-2 × m+1,j+m),
21                  i-m), (2 × m ≤ i ≤ N), (0 ≤ j ≤ i-2 × m);
22          end of main loop;
23      Y[i] := H[i], (0 ≤ i ≤ N);
24      end;
```

The actual function of this algorithm follows quite easily from the definition of $A^{(k)}(i,j)$ and $H^{(k)}(i)$ derived in the previous section. The statement in lines 18-19 determines the values of $H^{(k+1)}(i)$ from $H^{(k)}$ and $A^{(k)}$ with procedure P computing the summation

$$\sum_{j} A^{(k)}(i,j) \times H^{(k)}(j)$$

on some appropriate range for j. Similarly, the statement in lines 20-21 determines the values of $A^{(k+1)}(i,j)$ from $A^{(k)}$ with procedure P computing

$$\sum_{\ell} A^{(k)}(i,\ell) \times A^{(k)}(\ell,j)$$

on some appropriate range for ℓ. The main loop of the program is executed $\lceil \log_2 (N+1) \rceil$ times. This allows the determination of y(i) as

$$y(i) = H(i)$$

To complete our development we now consider the execution time and processor requirements of the algorithm. First we consider the processor requirements. Each invocation of the inner product procedure, procedure P, requires N+1 processors and for m = 1 there are about $N^2/2$ innter products to be computed. This yields $O(N^3)$ processors. For the execution time we note that each invocation of the procedure P requires $\log_2 (N+1)$ steps. This procedure is involved at most $2 \cdot \log_2 (N+1)$ times so the total time required is $O(\log^2 N)$.

To show that this algorithm is suited for execution on our machine, we consider a typical invocation of the procedure P. Consider the statement

$$A[i,j] := A[i,j] + P(A[i,*],A[*,j],...$$

We assign this computation to the set of PE's with labels (i,*,j). If we store A in memory in such a way that A(i,j) is in memory (i,j) then PE (i,j,k) will be able to access the necessary elements of A. The basic operations performed by this algorithm are the same as those of the matrix-matrix product algorithm except the limits on

$$\sum_j a_{ij} \cdot b_{jk}$$

are different. From these results the suitability of our machine for executing this algorithm can be seen.

IX. COMBINATORIAL APPLICATIONS

The applications that we have considered thus far have all been numerical in nature. As further examples of the versatility of our machine, we now consider two combinatorial problems, namely soring and permutations. The methods described are optimal in the sense of requiring minimal time.

In the case of sorting we consider the following problem. Given an array of n items, we want to rearrange these items into ascending order as quickly as possible. It can be shown, by an information theoretic argument, that the minimum number of comparisons required to do this sorting is $O(n \cdot \log n)$. This indicates that the best possible algorithm for a serial machine must require at least $O(n \cdot \log n)$ steps. For a parallel machine with n processors the minimum possible would be $O(\log n)$ steps, but the best known algorithm [Batcher, 1968] requires $O(\log^2 n)$ steps. In this section we describe a method for sorting that requires $O(\log n)$ steps when executed on the machine of Section II.

The strategy that we use for sorting is as follows. We first perform comparisons on all n^2 pairs of items. We assign a value of 1 to each comparison that indicates $<$ and a value of 0 to each comparison that indicates $=$ or $>$. Let c_{ij} denote the value assigned to the result of comparing the i^{th} element with the j^{th} element. The value of the summation

$$\sum_{i=0}^{n-1} c_{ij}$$

is the position of the i^{th} element of the original list in the sorted list if we count from 0 instead of 1 as the origin. This sum can be determined in O(log n) steps, using the standard log-sum algorithm, simultaneously for all n items under consideration.

To complete the sorting, we must now move the items into the correct location. The method used to do this part of the sorting procedure is also suitable for performing arbitrary data permutations. We proceed in the following manner. Make n copies of the entire vector under consideration. This is easily done on our machine in unit time. In the i^{th} copy of the vector we wish to select the element that is to end up in the i^{th} position of the output vector. To do this, in the i^{th} copy of the vector we zero out all entries except the one that is to end up in the i^{th} position of the output vector. After this modification, it is easily seen that the sum of all elements in the i^{th} copy of the vector is exactly the value of the element that is to end up in the i^{th} position in the output vector. To compute these sums we again apply the log-sum algorithm. This requires another O(log n) steps to perform the data permuting that is required.

Although this algorithm will work in the case where there are no duplications in the list to be sorted, a minor modification must be made to allow lists with duplications to be sorted. We make this modification in such a manner as to generate what is referred to in Knuth [1973] as a stable sorting algorithm, that is if there are items which have identical keys in the list to be sorted they appear in the same relative order in the sorted list as they did in the original list. We accomplish this by extending our comparisons in such a way as to eliminate the possibility of two items

being equal. In particular, we consider that

$$\text{if } X(i) = X(j) \text{ and } i < j \text{ then } X(i) < X(j)$$

That is to say an item is less than another item if its key is less or their keys are equal and it appears first in the original list.

Both the sorting and permuting methods use only n^2 PE's. Since there are n^3 PE's available this results in a very low level of resource utilization. It would be desirable if there were some method of make use of these idle PE's. From consideration of the nature of our machine and the operations required for the algorithm, it can be seen that the entire algorithm can be executed using only one block of n^2 PE's. If more than one permutation were to be done then it would be possible to perform up to n of them simultaneously by allocating a different block of PE's to each permutation. This would increase the resource utilization when applicable.

X. SUMMARY AND CONCLUSIONS

In this paper a machine architecture is presented that considers processing elements and memories to be independent resources. The architecture of such a machine is determined by the nature of the interconnections between the processing elements and the memories. By considering the mathematical formulation of a matrix-matrix product, a machine architecture is developed that allows computation of matrix-matrix products in $O(\log n)$ steps, where the matrices are $n \times n$. The machine has n^2 memories and n^3 processing elements. Stone [1973b] has given some results on the suitability of certain algorithms for parallel computation as related to questions of speedup and resource utilization. These results indicate that this machine would be reasonable to use for some classes of algorithms. Additionally, this architecture provides a valuable method for increasing the logical bandwidth of the machine memory without increasing the physical bandwidth in those cases where it is applicable.

We also developed an algorithm for the solution to triangular linear systems of equations that is as fast as the best algorithm known to this author. The algorithm requires $O(\log^2 n)$ steps for execution on the machine presented here. The other algorithm known to exhibit equivalent time behavior requires a MIMD computer with $O(n^4)$ processors for execution.

The development of our algorithm is based upon the principle of recursive doubling. In our proof we develop an extension of this principle

as compared to the work of Stone [1973a] and Kogge [1974]. This technique seems to be very valuable for introducing parallelism into problems which can be posed as recurrence problems.

At this writing the error behavior of the numerical algorithms in this paper has not been investigated. The work of Kogge [1972] indicates that the error behavior could be expected to be as good or better than that of the serial algorithms.

Sorting and permutations are two important combinatorial type problems. We have described methods for both of these problems that require only O(log n) steps when applied to vectors of n elements using our machine.

ACKNOWLEDGEMENT

The author expresses his appreciation to Professor Harold Stone of Stanford University for pointing out the paper by Heller [1973] which inspired this work and to Bell Telephone Laboratories for the financial support which made this work possible.

BIBLIOGRAPHY

Bell, C. G. and Newll, A., Computer Structures: Readings and Examples. New York, New York: McGraw Hill, 1973.

Flynn, M. J., "Very high-speed computing systems," Proceedings of the IEEE, Vol. 54, no. 12, pp. 1901-1909, December 1966.

Heller, D., "A determinant theorem with applications to parallel algorithms," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1973.

Knuth, D. E., The Art of Computer Programming, Vol. 3, Searching and Sorting. Reading, Massachusetts: Addison-Wesley, 1973.

Kogge, P. M., "The numerical stability of parallel algorithms for solving recurrence problems," Rep. 44, Digital Systems Laboratory, Stanford University, Stanford, California, September 1972.

Kogge, P. M., "Parallel algorithms for the efficient solution of recurrence problems," IBM Journal of Research and Development, Vol. 18, No. 2, pp. 138-148, March 1974.

Stone, H. S., "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," Journal of the ACM, Vol. 20, No. 1, pp. 27-38, January 1973a.

Stone, H. S., "Problems of parallel computations," Proceedings of the Symposium on Complexity of Sequential and Parallel Numerical Algorithms. New York, New York: Academic Press, 1973b.

Strassen, V., "Gaussian elimination is not optimal," Numerische Mathematik, Vol. 13, pp. 354-356, August 1969.

Winograd, S., "A new algorithm for inner product," IEEE Transactions on Computers, Vol. C-17, pp. 693-694, July 1968.

```
Definitions;
Index Variable i = {0,1,...,n-1}
Index Variable j = {0,1,...,n-1}
Index Variable k = {0,1,...,n-1}
Memory MEM[0:max_mem-1]; "This is the total memory of the system"
Register CUR[0:nreg-1]; "Control unit registers"
Register CU_CC; "Control unit condition code register"
Register PC; "Program counter"
Register PER[0:n-1,0:n-1,0:n-1][0:nreg-1]; "Processing Element Registers --
     PER(i,j,k)(m) is the m^th register in PE(i,j,k)"
Register ENABLE[0:n-1,0:n-1,0:n-1]; "Enable bits -- ENABLE(i,j,k) is the
     enable bit for PE(i,j,k)"
Register PE_CC[0:n-1,0:n-1,0:n-1]; "Processing element condition code --
     PE_CC(i,j,k) is the condition code for PE(i,j,k)"
Register R[0:n-1,0:n-1,0:n-1]; "Routing register -- R(i,j,k) is the
     routing register for PE (i,j,k)"
Register I[0:n-1,0:n-1,0:n-1]; "I(i,j,k) contains the value i"
Register J[0:n-1,0:n-1,0:n-1]; "J(i,j,k) contains the value j"
Register K[0:n-1,0:n-1,0:n-1]; "K(i,j,k) contains the value k"
End of Definitions;
```

```
"Control Unit instructions"

Functions;

cu_addr ← (cu_reg2 = 0 → disp;

          cu_reg2 ≠ 0 → disp + CUR(cu_reg2));

End of Functions;

Instructions;

"     LOAD     cu_reg1,disp(cu_reg2)                          Load"

LOAD := (CUR(cu_reg1) ← MEM(cu_addr));

"     STORE    cu_reg1,disp(cu_reg2)                          Store"

STORE := (MEM(cu_addr) ← CUR(cu_reg1));

"     ADD      cu_reg1,cu_reg2                                Add"

ADD := (CUR(cu_reg1) ← CUR(cu_reg1) + CUR(cu_reg2));

"     SUB      cu_reg1,cu_reg2                                Subtract"

SUB := (CUR(cu_reg1) ← CUR(cu_reg1) - CUR(cu_reg2));

"     MPY      cu_reg1,cu_reg2                                Multiply"

MPY := (CUR(cu_reg1) ← CUR(cu_reg1) * CUR(cu_reg2));

"     DVD      cu_reg1,cu_reg2                                Divide"

DVD := (CUR(cu_reg1) ← CUR(cu_reg1) / CUR(cu_reg2));

"     LOADR    cu_reg1,cu_reg2                                Load Register"

LOADR := (CUR(cu_reg1) ← CUR(cu_reg2));

"     CMP      cu_reg1,cu_reg2                                Compare"

CMP := (CU_CC ← CUR(cu_reg1) : CUR(cu_reg2));

"     JMP      cond,disp(cu_reg2)                             Transfer"

JMP := (cond∧CU_CC → PC ← cu_addr);

"     BCAST    cu_reg1,pe_reg2                                Broadcast"

BCAST := (ENABLE(i,j,k) → PER(i,j,k)(pe_reg2) ← CUR(cu-reg1));

End of Instructions;
```

```
"Processing Element instructions"

Functions;

pe_addr(i,j) ← (mode = 'X' → (disp +
                    (cu_reg2 = 0 → 0; cu_reg2 ≠ 0 → CUR(cu_reg2)) +
                    (pe_reg2 = 0 → 0; pe_reg2 ≠ 0 → PER(0,i,j)(pe_reg2)))
                    □J(0,i,j)□K(0,i,j);
                mode = 'Y' → (disp +
                    (cu_reg2 = 0 → 0; cu_reg2 ≠ 0 → CUR(cu_reg2)) +
                    (pe_reg2 = 0 → 0; pe_reg2 ≠ 0 → PER(i,0,j)(pe_reg2)))
                    □I(i,0,j)□K(i,0,j);
                mode = 'Z' → (disp +
                    (cu_reg2 = 0 → 0; cu_reg2 ≠ 0 → CUR(cu_reg2)) +
                    (pe_reg2 = 0 → 0; pe_reg2 ≠ 0 → PER(i,j,0)(pe_reg2)))
                    □I(i,j,0)□J(i,j,0));
d ← disp + (cu_reg2 = 0 → 0; cu_reg2 ≠ 0 → CUR(cu_reg2));
End of Functions;
```

```
Instructions;
"     LOAD,mode     pe_regl,disp(cu_reg2,pe_reg2)          Load"
LOAD := (ENABLE(i,j,k) → PER(i,j,k)(pe_regl) ← (mode = 'X' → MEM(pe_addr(j,k));
                                                mode = 'Y' → MEM(pe_addr(i,k));
                                                mode = 'Z' → MEM(pe_addr(i,j)));
"     STORE,mode    pe_regl,disp(cu_reg2,pe_reg2)          Store"
STORE := (mode = 'X' → (ENABLE(0,j,k) → MEM(pe_addr(j,k)) ← PER(0,j,k)(pe_regl));
          mode = 'Y' → (ENABLE(i,0,k) → MEM(pe_addr(i,k)) ← PER(i,0,k)(pe_regl));
          mode = 'Z" → (ENABLE(i,j,0) → MEM(pe_addr(i,j)) ← PER(i,j,0)(pe_regl)));
"     ADD           pe_regl,pe_reg2                        Add"
ADD := (ENABLE(i,j,k) → PER(i,j,k)(pe_regl) ← PER(i,j,k)(pe_regl) +
                                              PER(i,j,k)(pe_reg2));
"     SUB           pe_regl,pe_reg2                        Subtract"
SUB := (ENABLE(i,j,k) → PER(i,j,k)(pe_regl) ← PER(i,j,k)(pe_regl) -
                                              PER(i,j,k)(pe_reg2));
"     MPY           pe_regl,pe_reg2                        Multiply"
MPY := (ENABLE(i,j,k) → PER(i,j,k)(pe_regl) ← PER(i,j,k)(pe_regl)*
                                              PER(i,j,k)(pe_reg2));
"     DVD           pe_regl,pe_reg2                        Divide"
DVD := (ENABLE(i,j,k) → PER(i,j,k)(pe_regl) ← PER(i,j,k)(pe_regl) /
                                              PER(i,j,k)(pe_reg2));
"     LOADR         pe_regl,pe_reg2                        Load Register"
LOADR := (ENABLE(i,j,k) → PER(i,j,k)(pe_regl) ← PER(i,j,k)(pe_reg2));
"     CMP           pe_regl,pe_reg2                        Compare"
CMP := (ENABLE(i,j,k) → PE_CC(i,j,k) ← PER(i,j,k)(pe_regl) : PER(i,j,k)(pe_reg2));
```

```
"     SETE     mode,cond                                    Set Enable Bit"
SETE := (ENABLE(i,j,k) ← (mode = ' ' → cond∧PE_CC(i,j,k);
                          mode = 'A' → cond∧PE_CC(i,j,k)∧ENABLE(i,j,k);
                          mode = 'O' → (cond∧PE_CC(i,j,k))∨ENABLE(i,j,k);
                          mode = 'C' → ¬ENABLE(i,j,k));
"     LDE,mode      disp(cu_reg2,pe_reg2)                  Load Enable Bit"
LDE := (ENABLE(i,j,k) ← (mode = 'X' → MEM(pe_addr(j,k))<i>;
                         mode = 'Y' → MEM(pe_addr(i,k))<j>;
                         mode = 'Z' → MEM(pe_addr(i,j))<k>;
"     STE,mode      disp(cu_reg2,pe_reg2)                  Store Enable Bit"
STE := (mode = 'X' → MEM(pe_addr(j,k))<i> ← ENABLE(i,j,k);
        mode = 'Y' → MEM(pe_addr(i,k))<j> ← ENABLE(i,j,k);
        mode = 'Z' → MEM(pe_addr(i,j))<k> ← ENABLE(i,j,k));
"     ROUTE,mode    disp(cu_reg2)                          Route"
ROUTE := (R(i,j,k) ← (mode = 'X' → R(i,((n·j+k-d)/n) mod n,(k-d) mod n);
                      mode = 'Y' → R(((n·i+k-d)/n) mod n,j,(k-d) mod n);
                      mode = 'Z' → R(((n·i+j-d)/n) mod n,(j-d) mod n,k));
End of Instructions;
```