THE UNIVERSITY OF CALGARY

MATRIX ARITHMETIC ON A SYSTOLIC ARRAY PROCESSOR

by

Donald J. Colpitts

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL ENGINEERING

CALGARY, ALBERTA
OCTOBER, 1985

ISBN 0-315-29945-2

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Matrix Arithmetic on a Systolic Array Processor" submitted by Donald James Colpitts in partial fulfillment of the requirements for the degree of Master of Science.

Dr. G.S. Hope (Supervisor)
Department of Electrical Engineering

Dr. O.P. Malik
Department of Electrical Engineering

Dr. L.E. Turner
Department of Electrical Engineering

Dr. G.M. Birtwistle
Department of Computer Science

Date: March 7, 1986

ABSTRACT

In this thesis a set of algorithms is developed to evaluate matrix equations on a two dimensional systolic array processor. The array consists of programmable processing elements interconnected in a fixed configuration. Data and instructions flow through the array as wavefronts to allow the execution of all of the algorithms to be overlapped. Software simulation is used to verify the correct operation of the algorithms and the array processor.

ACKNOWLEDGEMENTS

The author wishes to express thanks to his thesis supervisor, Dr. G.S. Hope, for providing guidance during this work, and to Dr. G.M. Birtwistle for putting together the framework from which the simulation work evolved.

The friendliness and helpfulness of the other graduate students in the University of Calgary is appreciated. Without the encouragement of family and friends this work could not have been completed.

The financial support of the Natural Sciences and Engineering Research Council of Canada is acknowledged.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

DEMOS	Discrete Event Modeling on Simula
GE	Gaussian Elimination
LU Decomposition	Lower-Upper triangle Decomposition
$O(n)$	time complexity of order n
$O(n^3)$	time complexity of order n^3
PE	Processing Element
PPIV	Partial Pivoting
TPIV	Total Pivoting
W,N,S,E,NW,NE,SW,SE	compass directions relative to a PE
alu	arithmetic logic unit for PE
para	parameter storage memory for PE
q1,q2	queue pointers for PE
ram	matrix memory for PE
sp	stack pointer for PE

Chapter 1

THE GOAL OF THIS RESEARCH

This research is aimed at developing a set of algorithms to perform the arithmetic needed to evaluate matrix equations using a systolic array processor in real time..

1.1 DEFINITIONS

The above statement contains a number of important keywords and phrases which demand definition and explanation.

Algorithm

The dictionary defines an algorithm as "any special method of solving a certain kind of problem". Horowitz [1] defines the term as it applies to computers as:

> "An algorithm is a finite set of instructions which, if followed, accomplish a particular task. Every algorithm must satisfy the following criteria:
> INPUT: there are zero or more quantities which are externally supplied;
> OUTPUT: at least one quantity is produced;
> DEFINITENESS: each instruction must be clear and unambiguous;
> FINITENESS: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
> EFFECTIVENESS: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, it must also be feasible."

Set of Algorithms

This phrase implies a similarity between the algorithms with respect to input, output and the principles upon which the algorithms are based. It also suggests that the operations of the different algorithms are compatible.

Matrix Equation Arithmetic

In general, a matrix equation may involve at least the following operations:

Matrix Scaling by a Constant

Matrix Addition and Subtraction

Matrix Multiplication

Matrix Transposition

Matrix Inversion.

Furthermore, the arguments for these operations may involve constants, vectors, rectangular matrices or square matrices.

Systolic

Systole, the noun form of the adjective "systolic", refers to the rhythmic contraction of the heart that forces blood through the body. A systolic system is one that functions on the principle of having a regular and rhythmic flow to sustain it. In the case of the human body, blood is the sustaining substance, whereas it is the data needed to carry out calculations that sustains a computer.

The systolic concept, in terms of computer architecture, was developed

at Carnegie-Mellon University (circa 1978) by H.T. Kung [2]. Synchronization of a systolic system is provided by the rhythmic flow of data through the system. Data flows into the system in regular patterns such that the data is in the right places at the right times. These properties make systolic systems relatively simple to design.

Array Processor

An array processor is a group of similar, if not identical, processors that perform the same set of tasks on different streams of input data.

A manual telephone switchboard received so many calls that some callers had to wait unreasonably long times before their connection was made. How could this overloaded system be modified to handle the number of calls? One solution is to train the switchboard operator to be quicker at connecting calls. Unfortunately, human performance does have limits, so eventually even the fastest operator will not be able to keep up to the demand. Another solution is to hire a second operator. Two operators should be able to handle twice as many calls and this solution has other advantages:

-both operators perform the same set of tasks, so that they can be taught using the same training program,

-the solution extends to the almost inevitable situation that in a couple of years, when the number of calls again becomes too large, a third operator can be hired,

-the operators are interchangeable, so if one operator is sick, another operator can take over.

This analogy represents the situation occurring in many areas of data processing. Semiconductor technologies are being pushed to their performance limits, yet more and more applications require higher performance. Fortunately, many of these applications involve a limited set of tasks and can be satisfied by increasing the number of data processors.

Real Time

In the telephone switchboard analogy, the scenario was presented where some callers experienced undesirably long delays in having their calls serviced. In some applications it is necessary to process data before new data destroys the old data. Even if the old data is protected from being overwritten, the average rate at which data is processed must not be less than the input rate of new data if the delay in processing the data is to remain bounded. Such an application involves real time constraints, and the system on which the data is processed operates in real time. Real time applications put an emphasis on system performance.

Putting all these definitions together again: the goal of this research is to develop a set of algorithms to perform the operations needed to evaluate matrix

equations on a systolic array processor in real time.

1.2 THE STRUCTURE OF THIS THESIS

The design of this parallel processing system begins in Chapter Two with an examination of the constraints imposed by a VLSI implementation. These constraints lead to an array processor configuration that only allows adjacent processing elements (PEs) to communicate. The number of different kinds of PEs is limited by the desire to use regularity and repetition to reduce the complexity of the array.

A set of algorithms is developed in Chapter Three to implement the desired matrix operations on the loosely defined array architecture. These algorithms are based on the wavefront concept that regulates the flow of data, instructions and results through the array.

Chapter Four describes how the PEs and the array are modeled in order to verify the operation of the algorithms and estimate their performance. The application of the algorithms and the array to the evaluation of the Kalman filtering equations gives an indication how well the objectives are met.

The findings of this research are summarized in Chapter Five, along with suggestions regarding further work in the area of parallel algorithm and array processor design.

Chapter 2

THE CONCEPTUAL FRAMEWORK

2.1 PARALLELISM: POWER AND PERIL

Parallelism in computing refers to the presence of a similarity in the types of calculations to be performed. Often the only difference between calculations is the data used as operands. A system of processors can exploit this parallelism by having each of the similar calculations performed concurrently on an individual processor.

2.1.1 Power: The Motivation for Parallelism

As computers become more powerful, a larger number and variety of applications become feasible. Tremendous increases in the capacity of computers that have taken place in the past have undoubtedly caused the emergence of computer technology in fields as diverse as medicine, education, communication, control systems, and entertainment. It has been called the "Computer Revolution".

However, further advances in the power of computers will be due to different factors than in the past. Several important conclusions were reached at a recent conference on solid state circuits [3]:

-over the last decade integrated circuits (ICs) have become more

powerful largely due to their decrease in size,

-IC miniaturization is nearing the limit beyond which there may be more problems than benefits,

The reasons for this prediction are:

-a scaled-down transistor can operate proportionately faster, but the time delay associated with a scaled down interconnect does not decrease [4], and now the point has been reached where interconnects limit the speed of circuits,

-beyond present levels of miniaturization the decrease in circuit size due to scaled down devices is exceeded by the increase in chip area due to larger numbers of interconnections,

-limitations in the number of pins available using present packaging methods has become restrictive in terms of the complexity of the circuit that can be put into a single package.

The decreased potential for improvement in the performance of ICs by miniaturization has sparked increased interest in the use of parallel processors to give improved system performance. The motivation, as suggested in the "telephone switchboard" analogy, is that extending the mostly heavily utilized component in a system often results in an increase in system capacity. Because many applications are computationally intensive, it is frequently the processing unit inside the computer that is the limit to the speed at which applications can be

processed. Parallel processing increases the capacity of a computer system by exploiting the parallelism inherent in many applications. Figure 2.1 shows a simplified view of the recent trend toward improving performance through parallel processing. Notice that parallel processing adds another dimension to the quest for greater system performance. The use of parallelism is applicable to both the hardware and to the software that comprise a system.

H.T. Kung [5] discusses the need to balance processing power with input/output (I/O) capacity in order to avoid overkill in a computer design. Increasing the computational capacity of a computer may result in some other aspect of the operation of the computer limiting its speed.

The next sections examine the principles by which parallel processing systems are designed and some of the problems that must be overcome.

2.1.2 Approaches to Achieving Parallelism

Systolic architectures belong to the broad class of parallel processing architectures. Some characteristics of systolic machines are shared with other types of architectures such as data flow and control flow machines. Lerner [6] discusses data flow and control flow architectures in detail.

Control flow machines execute a single sequence of instructions; all of the processors do the same instruction at the same time. The term "single instruction, multiple data path" (SIMD) is commonly used to classify this type of machine

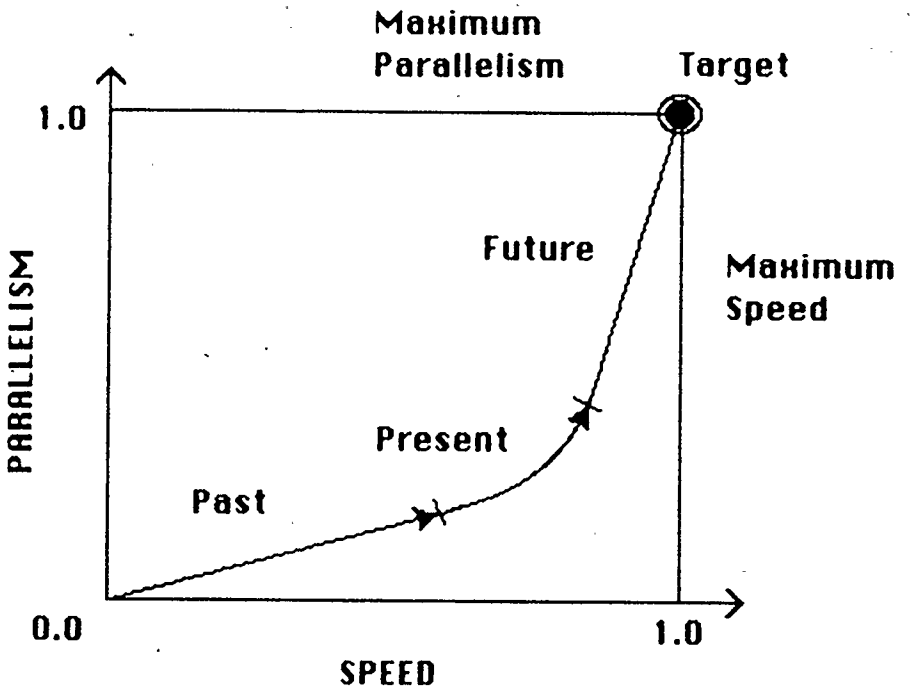


Fig. 2.1: The trend toward using parallelism to improve system performance.

[7]. Systolic machines frequently have all their processors performing the same sequence of instructions, but unlike control flow machines, adjacent processors usually do not execute the same instruction concurrently. For example, each processor might work on a different recursion of the same recursive algorithm.

Data flow machines involve a group of processors that each execute their own sequence of instructions when the data needed for the operations become available. Systolic machines make use of this same principle. As H.T. Kung [5] points out however, classical data flow machines involve a linear arrangement of processors that pass data in one direction, at a single speed, with only results being passed between processors. In contrast, systolic architectures often involve two-dimensional configurations of processors that may pass data in several directions, at different speeds and often pass inputs and intermediate results ,as well as final results, between processors.

Thus, systolic systems can be thought of as parallel architectures that are distributed in space and in time. An input piece of data is passed along between the processors that require its value. Each of these processors may calculate results or partial results based on the input data and may pass these results along to other processors that can make use of them. Eventually, the results of the collective effort of the processors emerge from the system.

For the flow of data and results to be rhythmic, a suitable configuration of processors and a corresponding format for inputting data is required such that

processors that share data are near each other so that the flow of data is orderly.

Since the development of the systolic concept, it has been shown that systolic systems can be designed to perform a large variety of tasks. H.T. Kung [5] gives a sampling of some of the types of operations performed by systolic systems. These include applications in signal processing, matrix multiplication and pattern matching. New applications for this powerful concept are appearing frequently.

2.1.3 Peril: Managing Chaos

With all the potential speed and power available in parallel processing comes the need for the designer to institute simplicity and regularity if order is to be maintained. This section identifies some of the problems associated with the use of parallelism and builds up a set of guidelines for keeping the design of parallel systems manageable.

2.1.3.1 Computational and Memory Access Bottlenecks

Computers and computer algorithms tend to be limited in speed by the time needed to carry out calculations and the time needed to access data from where it is stored. Generally one of these factors dominates the other.

H.T. Kung [5] defines computationally bound algorithms as those involving more computations than pieces of data, including both input data and

results. Accordingly, an input/output bound algorithm is one that is not computationally bound.

Multiplication of matrices is computationally bound since each element of each matrix is used several times as an input for the calculation of the elements in the product matrix. Addition of matrices is I/O bound since each element of each matrix is used only once as an input for the calculation of the elements in the sum matrix.

Parallel processors exploit the independence of calculations inherent in many computationally bound problems to reduce the time needed to compute results. Reduction of the memory access bottleneck can be achieved by overlapping input and output with the calculations, by avoiding repeated memory accesses for the same piece of data, and by smoothing out the demand for input data. Systolic computer designs and systolic algorithms make use of all these ideas.

2.1.3.2 The Effect of Interconnects on Circuit Design

The cost of interconnects in terms of layout area and time delay must be taken as an important criteria in formulating circuit designs. Sutherland and Mead [8] illustrate how the circuit area devoted to interconnections grows more than linearly with circuit complexity because larger circuits require a greater number of connections to span a larger range of distances. This leads to the problem mentioned by Fischetti [3], that as devices and interconnects are miniaturized

beyond a certain point, the amount of extra circuitry that can fit into the same area does not increase proportionately because of the increase in circuit area required for interconnections.

No longer are switching speeds of devices the limit to speed. The delay in communication along interconnects has become as significant.

The cost of interconnections in terms of speed and fabrication area is therefore an important issue in computer design. This cost is reduced by using small circuits layed out such as to minimize interconnect lengths. Regular patterns of interconnects tend to result in shorter interconnects as well as reduce layout complexity.

2.1.3.3 Processor Communication and Contention

Communication between a group of strongly connected processors can be very complex. The possibility of more than one processor wanting to communicate with the same processor simultaneously, results in the need for complex protocols to escape an otherwise contentious situation.

These concerns are particularly relevant to parallel processing systems which, unlike single processor systems, must contend with problems in the communication of information between processors. Sutherland and Mead [8] support this view:

> "The challenge in designing or using a parallel processor of any of these three types [pipelined processors, array of processors, distributed processors] lies in discovering ways which simple patterns of communication within the [system of] processors can be made to match the communication tasks inherent in the problem being solved."

Systolic systems tend to be designed to cope with a particular family of algorithms in mind. Often this similarity extends to the patterns of communication needed to perform the algorithms. Introducing regular and repeated patterns of communication amongst processors simplifies the design of the system. Reducing the number of connections between processors greatly reduces communication conflicts as well as reducing the complexity of the layout of the IC or circuit board on which the system is fabricated.

2.1.3.4 Asynchronous versus Synchronous Operation

Clock skew is caused by having signals travel along wires or interconnections of different lengths and by physical variations along the wires themselves. Synchronization failure can result when processors receive clock signals along different paths. The time required to charge a clock line becomes prohibitively large as the lengths of the clock lines increase. The slowing down of the speed of operation of the system to account for these factors reduces system performance. For large systems, it is necessary to use alterative clocking schemes or design systems and algorithms that do not rely on global synchronization.

Fischer and H.T. Kung [9] note that arbitrarily expandable two dimen-

sional arrays are not feasible using synchronous clock schemes. S.Y. Kung and Gal-Ezer [10] also examine the timing of systolic arrays. By replacing the global clock with a protocol for communication between adjacent processors, a synchronous system is changed to an asynchronous self-timed system that makes use of the data flow principle. Calculations proceed to the rhythmic beat of data moving through the system rather than to the beat of global clock. The advantage of the asynchronous system is the elimination of the global communication of timing signals that costs so much in performance, area and layout complexity.

2.2 SUMMARY

Design recommendations aimed at reducing some of the problems discussed above include making use of local communication and using regular geometries and repetitive structures as opposed to random designs. These guidelines apply to both the design of the computer system and to the design of the algorithms which the system will perform. The next chapter considers architectures and algorithms that adhere to these recommendations.

Chapter 3

WAVEFRONT ALGORITHMS FOR MATRIX OPERATIONS

3.1 THE CONCEPT OF A WAVEFRONT

The wavefront concept was developed by S.Y. Kung et al [11]. Like the systolic concept, it depends on the locality, regularity, recursiveness and concurrency of certain algorithms. But it extends the systolic concept by employing the particularly illustrative analogy of the flow of data to the movement of wavefronts.

The next sections identify the properties of wavefronts and define the rules that wavefronts must obey.

3.1.1 The Composition of Wavefronts

The notion of a wavefront is used to indicate a similarity in the type of information in the array of PEs at a particular time. The idea of joining points which contain similar information or that are performing the same task is like the use of contour lines on topographic maps or isobars on weather maps. The lines provide visual identification of areas which share a common property. The directions in which the wavefronts move indicate the directions of flow of information within the array.

A wavefront can be comprised of constants, as in the case of a parameter that is communicated to all PEs. A wavefront can be a call to a procedure which implements a matrix operation such as matrix addition or matrix multiplication. Wavefronts can represent variables such as the sum of the products of pairs of data used to calculate the product of two matrices. Finally, a wavefront can represent a specific step in a calculation, such as a recursion in a recursive algorithm. Whatever abstraction is attached to a wavefront remains with that wavefront from the time the wavefront enters the array, to when the wavefront leaves the array. This allows complicated operations to be broken down into a series of wavefronts representing instructions, parameters, data, variables and recursions.

3.1.2 The Source of Wavefronts

Wavefronts always originate from a designated source. As a wavefront moves through an array of processors, each PE acts as a secondary source of the wavefront perpetuating its movement and direction. This means that wavefronts that come from the same source will travel parallel to one another. The source of wavefronts can move around in the array provided that the distinction of being "source" is only passed between adjacent PEs. This insures that wavefronts from a moving source will never intersect. Conceptually, intersecting wavefronts represent a contentious situation. The rule that wavefronts cannot proceed until the receiving PE is ready to accept the wavefront removes the opportunity for wavefronts to overtake one another.

Instruction wavefronts flow into the array from the northwest. Therefore, it is required that all instructions begin and end with PE(1,1) as the source. In figure 3.1, input data is shown entering the array from the north and from the west. Output data leaves the array from the south and the east. Diagrams of this form are used to illustrate the arrangements of data and instructions needed to perform different matrix operations.

Many instructions require that parameters be communicated to all the PEs. Figure 3.2 shows the pattern of communication of instructions and parameters between PEs that creates wavefront motion parallel to the main diagonal of the array. This pattern extends to larger or smaller arrays and is also applicable to non-square arrays. The rules that summarize this pattern are:

-data received on one side of the PE is passed out the other side,

-data can enter or leave the array only at the edges of the array,

-PEs along the main diagonal of the array pass data in three directions in the order shown.

The sequence of diagrams in figure 3.3 illustrates the propagation of instruction wavefronts as they are passed along between PEs.

Instructions, parameters and data for a particular matrix operation each take up a specific number of wavefronts. The wavefronts representing the next operation can immediately follow a previous set of wavefronts. This pipelined

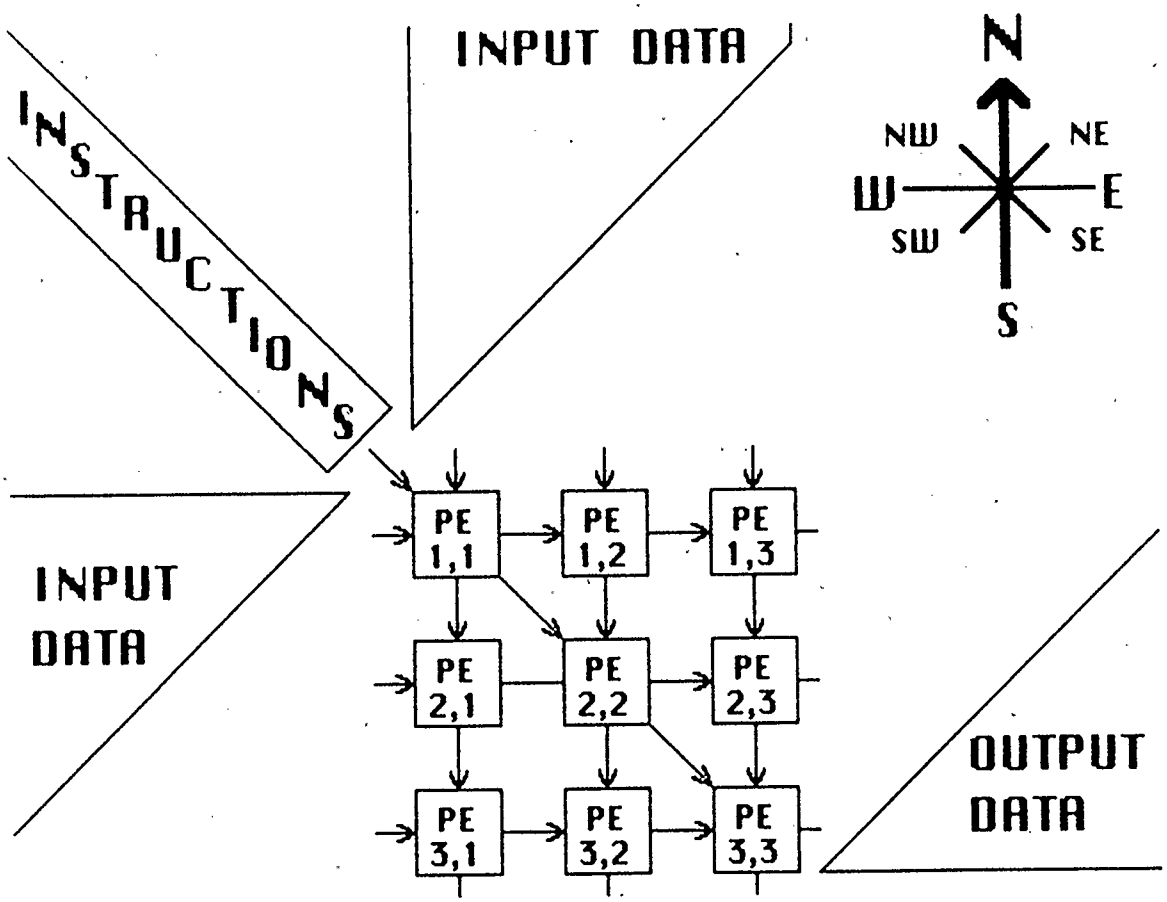


Fig. 3.1: Direction of flow of data and instructions.

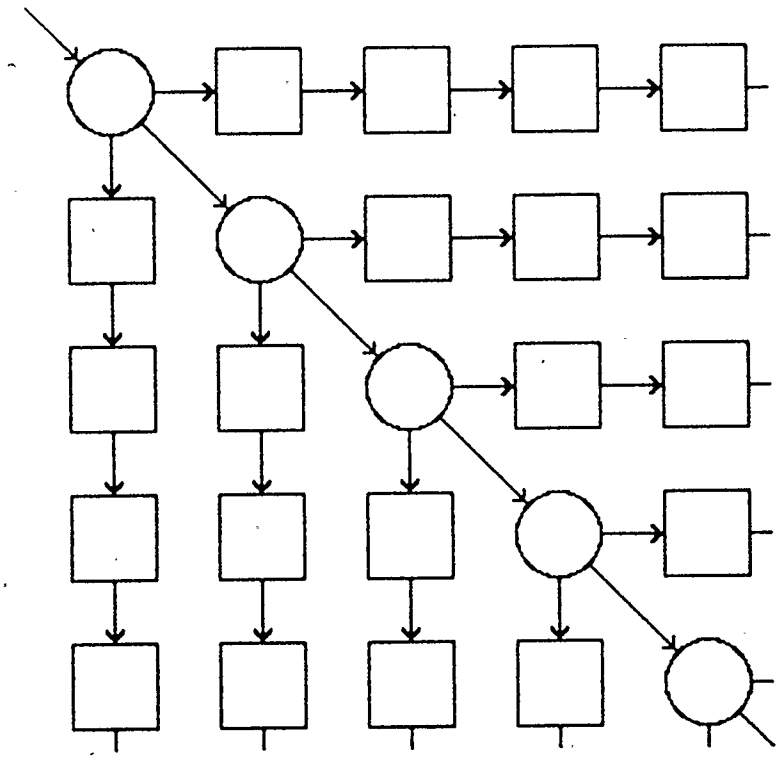


Fig. 3.2: Communication pattern for a wavefront array processor.

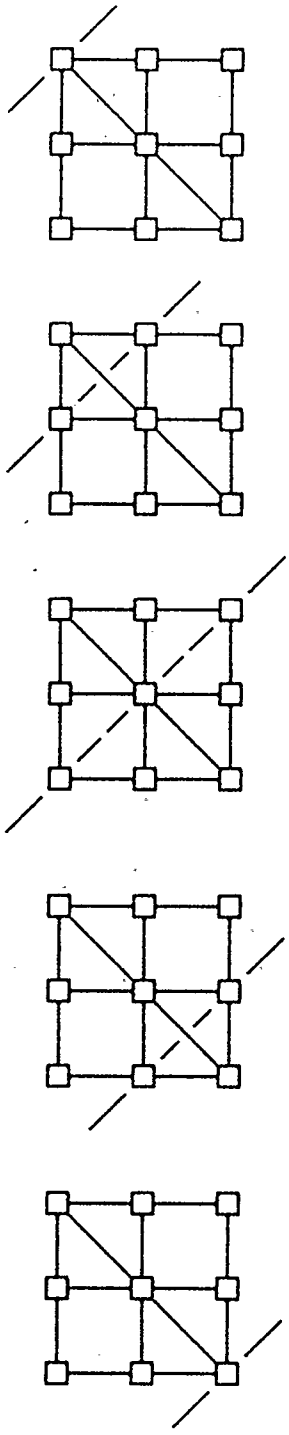


Fig. 3.3: Wavefront movement through an array of PEs.

flow of operations through the array is shown by the example situation in figure 3.4.

Frequently, the PE in the northwest corner of the array, PE(1,1), is the source of wavefronts. However, occasionally the source may move further within the array. The patterns of wavefronts resulting from these situations are shown in figure 3.5. Notice how the pattern can look like ripples in a pond, or like breakers approaching a beach depending on the location of the source, but the waves always move parallel to each other.

The concept of wavefronts allows the designer to visualize what is happening in the array of processors. This analogy is used extensively to describe the flow of data and instructions in the sections dealing with the design of algorithms for matrix operations.

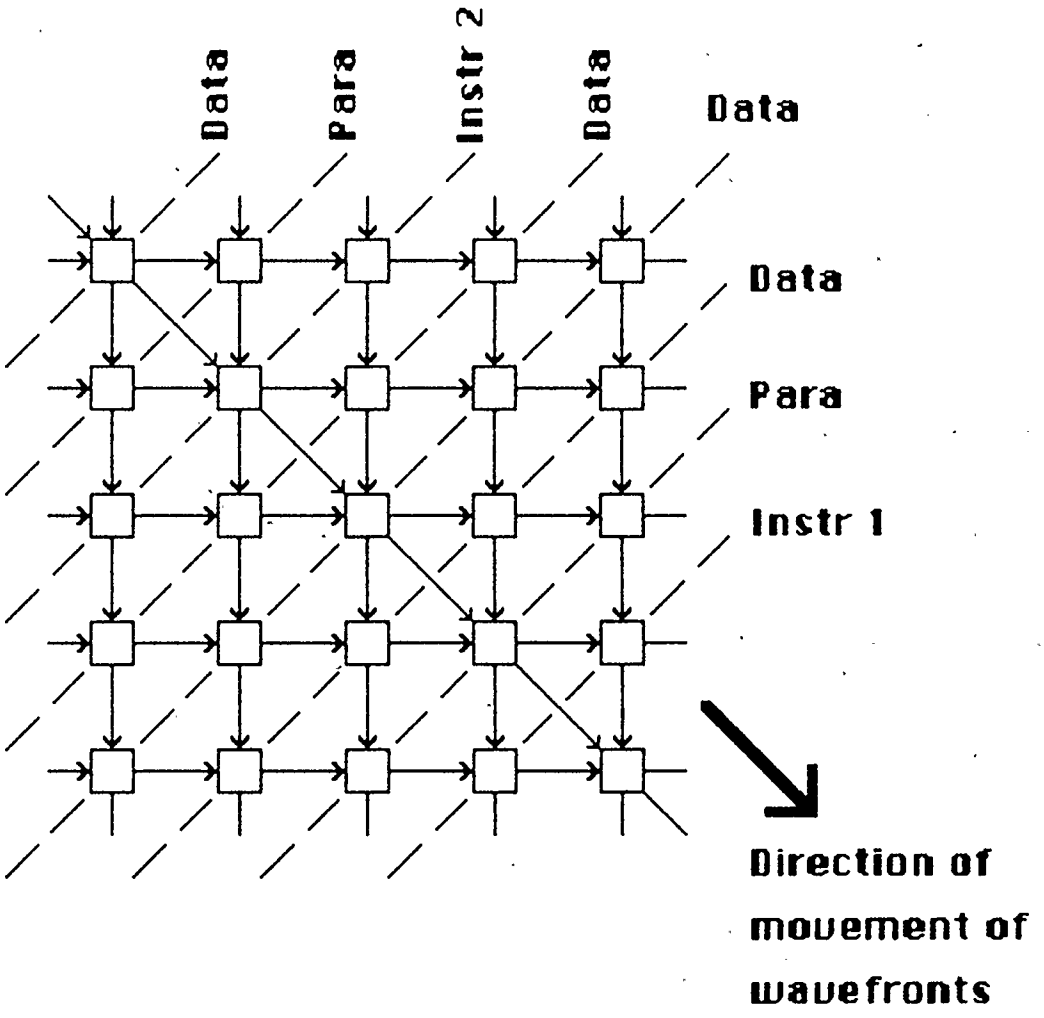
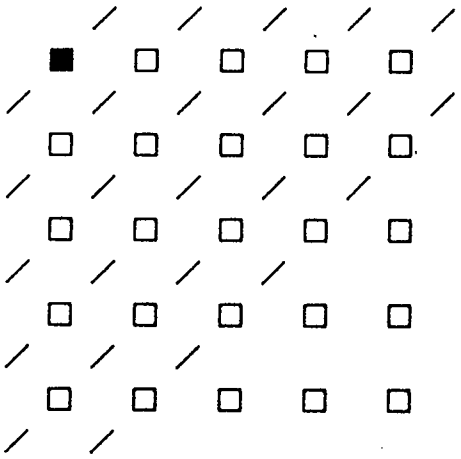
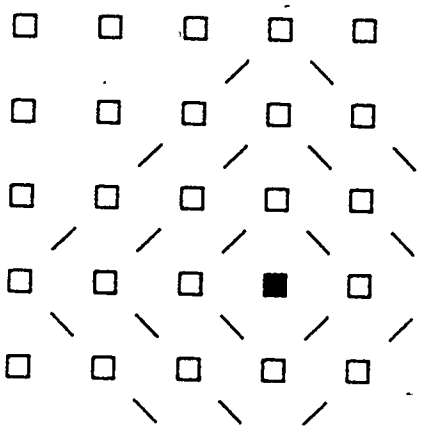


Fig. 3.4: Multiple operations passing through an array.



a) Source in the northwest corner of the array.



b) Source inside the array.

Fig. 3.5: Wavefront patterns generated by different sources."

3.2 LOADING AND UNLOADING THE ARRAY PROCESSOR

3.2.1 The Need for Loading and Unloading

Neither the load nor the unload operation accomplishes any computation, however, both are needed to give the array processor instruction set the flexibility to efficiently evaluate matrix equations.

Many of the algorithms designed for systolic array processors flow input data into the array while calculations are being performed. However, there are occasions when it is desirable or necessary to load a matrix or matrices into the array before calculations begin. Both the matrix addition and the Gaussian elimination algorithms require loading of matrix data prior to the execution of the calculations.

Similarly, algorithms which perform matrix scaling and matrix addition finish with the results of the calculations residing in the array. If these results are final results, then it is necessary to unload them from the array. If these results are only intermediate results in a more complicated calculation, then it is not necessary to unload them. In fact, it is possible to increase the speed of complicated calculations by keeping intermediate results in the array thus eliminating the need to reload them later.

The uses of LOAD and UNLOAD operations with computational operations are described in detail in the sections that deal with algorithms for

specific types of matrix calculations.

3.2.2 Loading One Matrix into the Array

The array of PEs is loaded using a simple technique. Each PE keeps the first piece of data it receives. All other data is passed along to the next PE. This technique is summarized in figure 3.6. The set-up for the LOAD operation is shown in figure 3.7. Notice that the LOAD instruction wavefront and all the necessary parameters precede the data wavefronts through the array.

The sequence of steps that comprise the loading of a three by three matrix is shown in figure 3.8. This sequence shows how the matrix being loaded into the array is confined to three wavefronts. Each wavefront represents a column of the matrix. As wavefront j passes through column j of the array, the matrix elements of column j are left in the appropriate PE. At the end of the sequence, each PE(i,j) contains matrix element a(i,j).

A matrix can be loaded into the north side of the array using the same technique by flowing the data south rather than east.

3.2.3 Loading Two Matrices into the Array

Two matrices can be loaded into the array by performing a LOAD across the array and a LOAD down through the array at the same time. Each PE stores the first piece of data that it receives from its north and west neighbors.

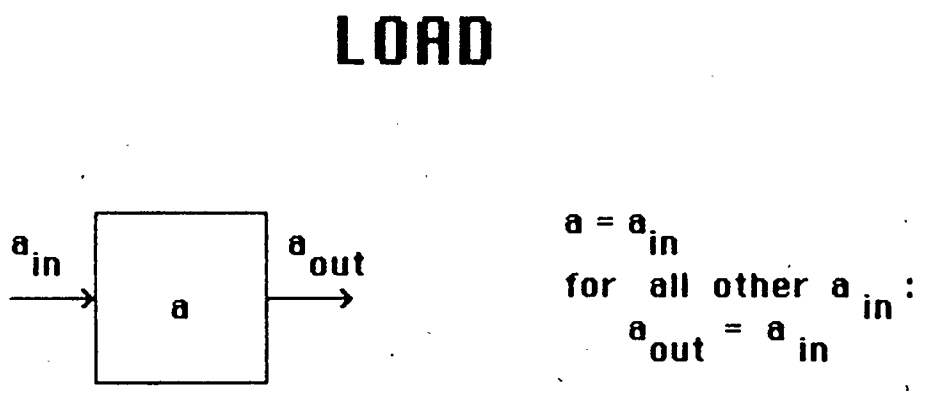


Fig. 3.6: Summary of the LOAD algorithm.

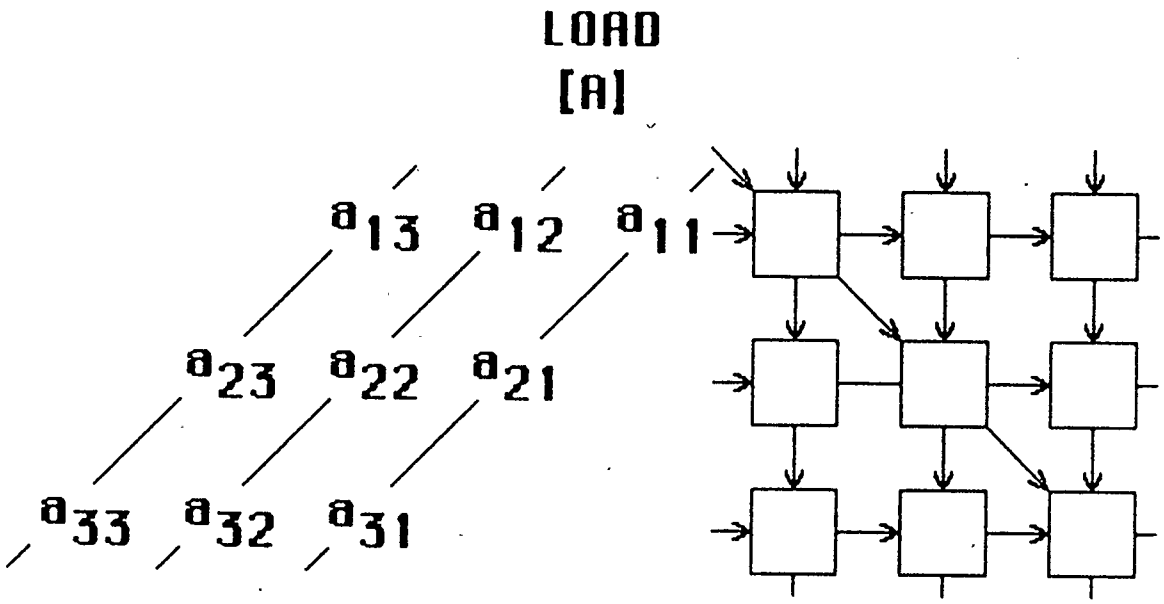


Fig. 3.7: Arrangement of data and instructions for LOADing.

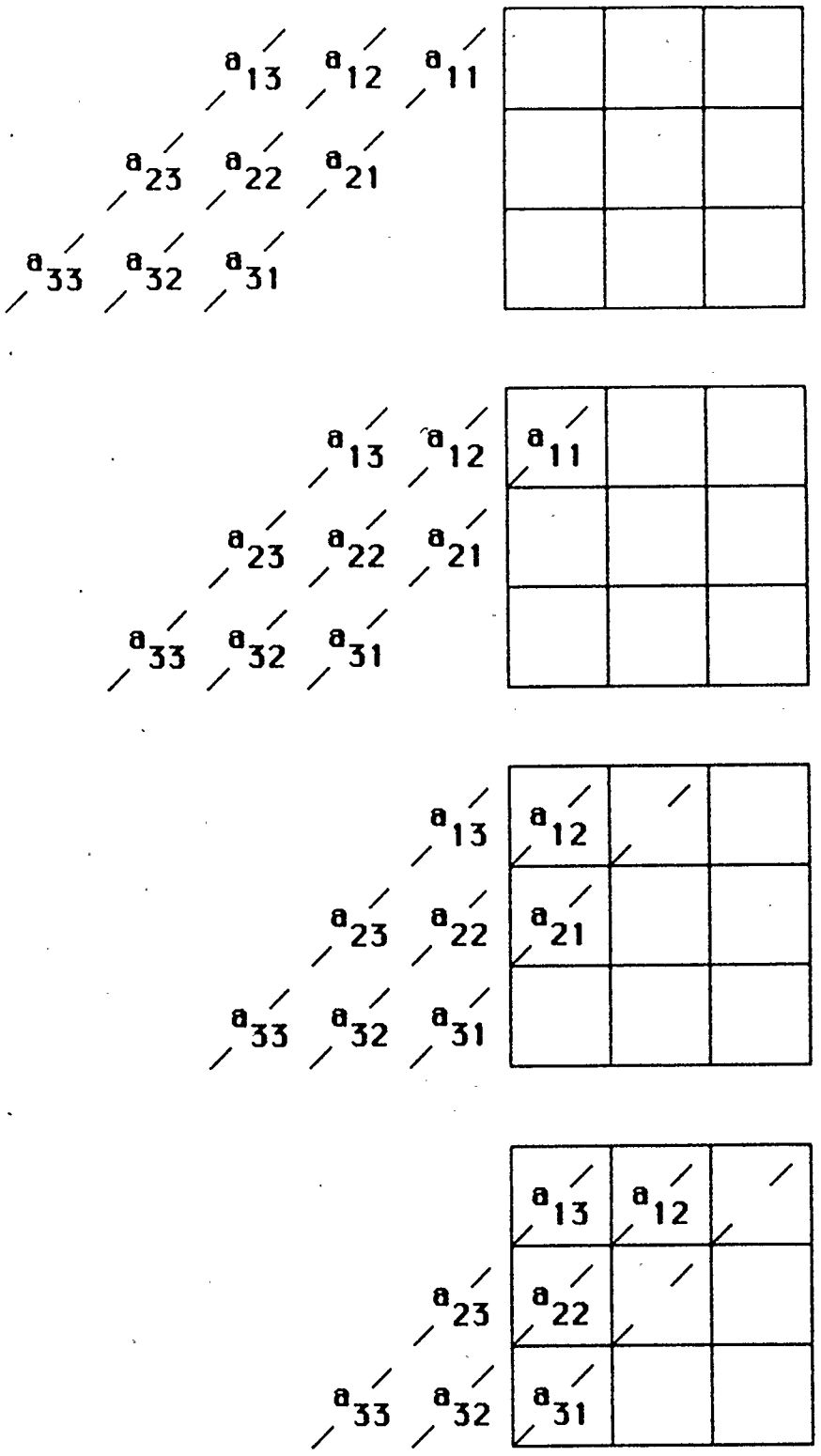


Fig. 3.8: Sequence of steps for loading a matrix into the array.

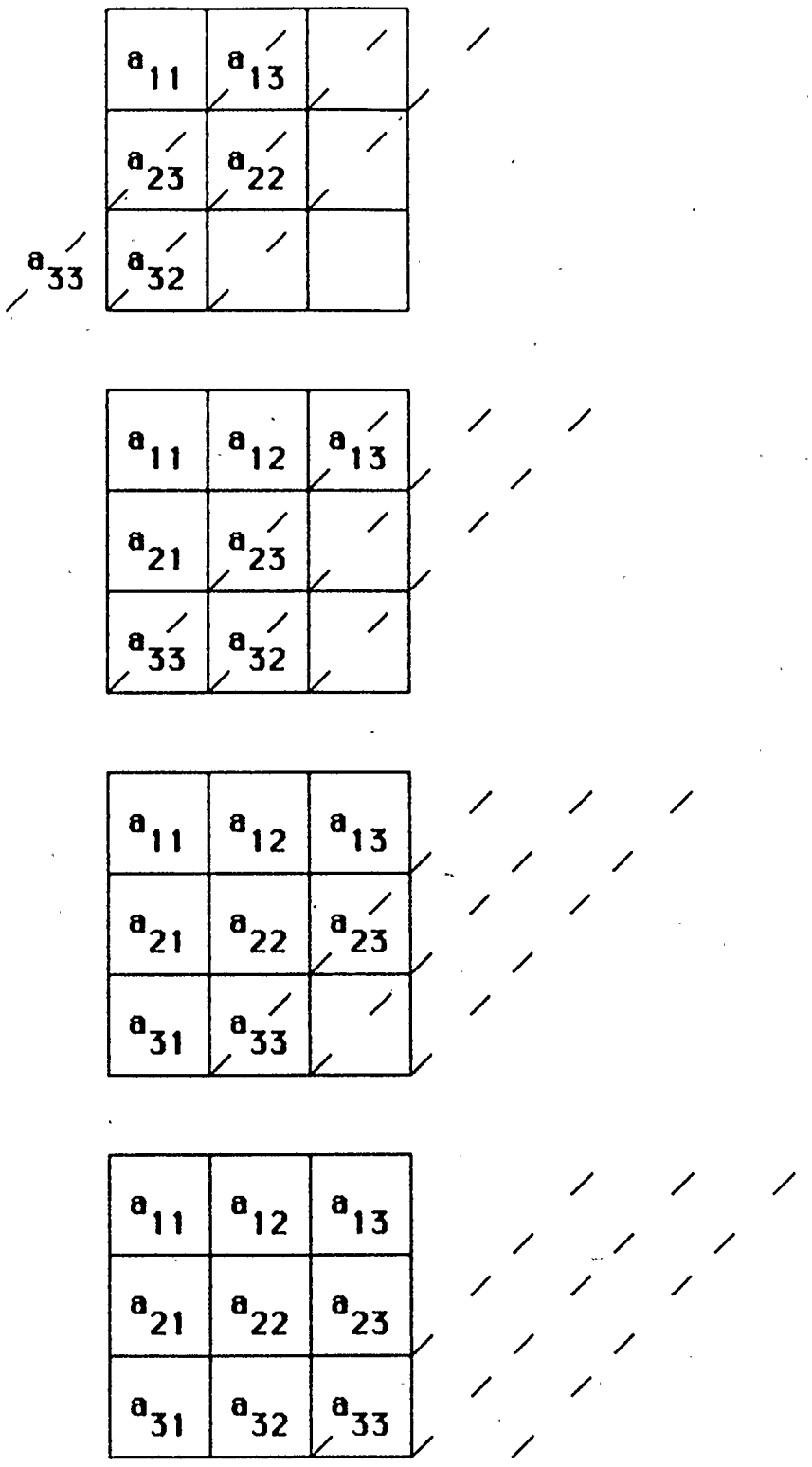


Fig. 3.8: (continued).

This requires that PEs have at least two temporary storage locations. All subsequent data is passed along in the same direction. This process takes the same amount of time as LOADing just one matrix, but each PE must do more work during each step. The step-up for LOADing from both directions is shown in figure 3.9. Notice that the arrangement of matrix B as it enters the array is the transpose of the arrangement of matrix A.

3.2.4 Unloading a Matrix from the Array

The pattern of data flow for unloading a matrix from the array is very similar to the pattern used to load the array. Each PE passes along all the matrix elements passed to it and appends its own matrix element's value to the end of the stream of data. This method is summarized in figure 3.10.

The arrangement of data in the array before the UNLOAD operation is identical to the arrangement of data after the LOAD operation. Also, the ordering of the matrix elements as they leave the array is the same as the order in which they were flowed into the array. Figure 3.11 shows the format of a matrix after it has been UNLOADed from the array.

3.2.5 Matrix Transposition using LOAD and UNLOAD

Matrices can be transposed by altering the order in which they are LOADed into the array. This corresponds to a change in the order of retrieval of the matrix elements from the external memory where they are stored. Similarly,

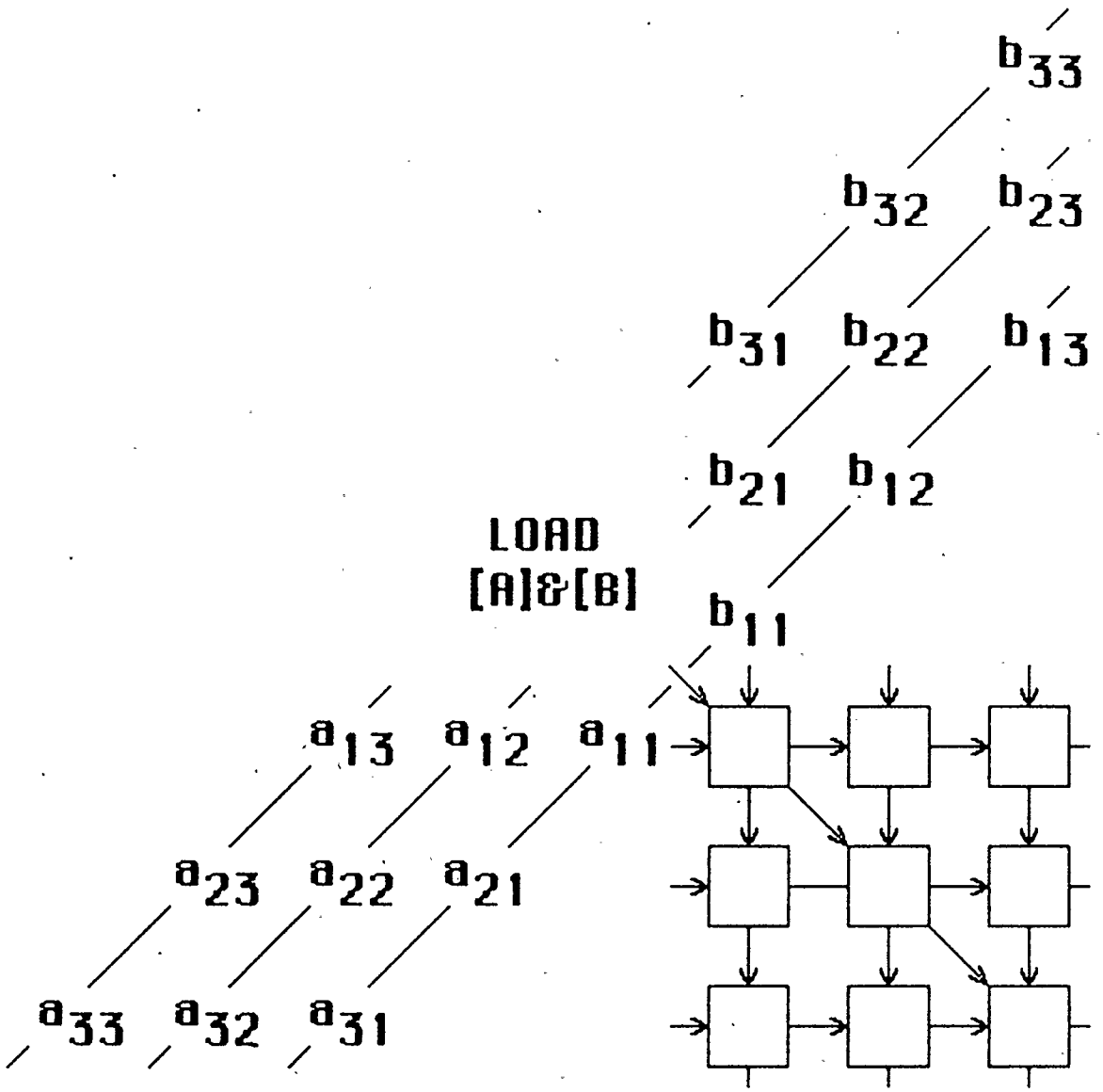


Fig. 3.9: LOADing two matrices simultaneously.

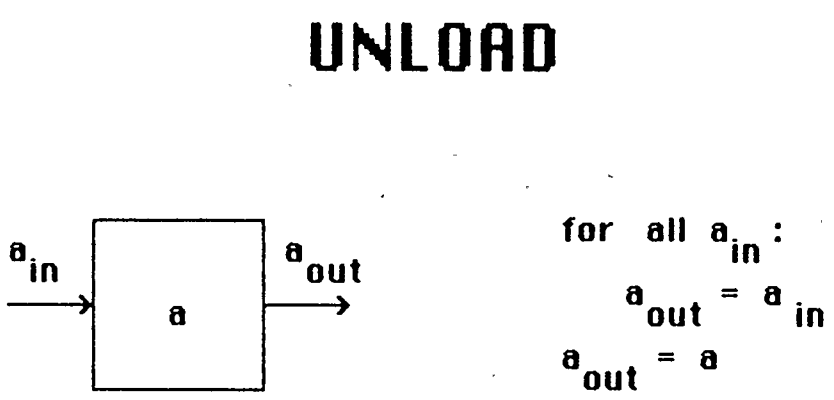


Fig. 3.10: Summary of the UNLOAD algorithm.

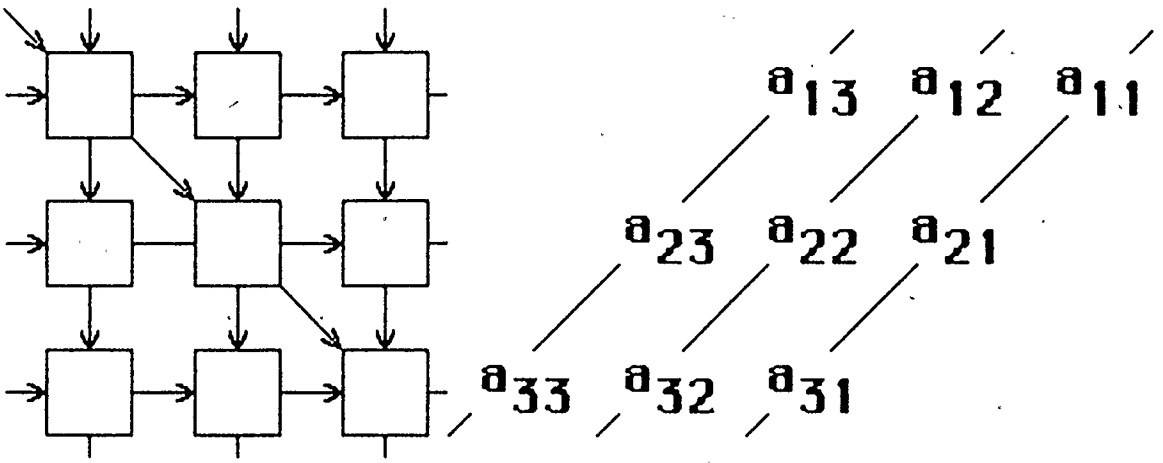


Fig. 3.11: Arrangement of matrix that has been UNLOADed.

matrices can be transposed as they are UNLOADed from the array and stored in the external memory. Transposition of a matrix residing in the array is not easily performed nor is it often necessary.

3.3 ADDITION, SUBTRACTION AND SCALING OF MATRICES

Matrix addition has been totally overlooked in literature dealing with systolic array processors. This is because adding matrices is not a computationally bound operation, thus, little is gained through increased computational capacity. On the other hand, matrix addition is considered here because the instruction set of a system designed to evaluate matrix equations must be complete.

The next section shows how matrix addition can be implemented on a wavefront array processor. All of the matrix addition techniques apply to subtraction of matrices as well.

3.3.1 Using LOAD and UNLOAD to Add Matrices

Because matrix addition is limited in speed by the I/O capacity of a system, the design of an efficient addition algorithm requires overlapping of I/O with calculations. It is desirable to have an algorithm which is flexible enough to handle cases where one or both of the matrices are already in the array of processors, possibly as intermediate results of some other operation. It is also advantageous to be able to leave the sum of two matrices in the array, as an intermediate result, or to be able to flow the sum out of the array if it represents a final result.

The most direct approach to matrix addition on a systolic array processor is simply to load the two matrices into the array, add corresponding elements and then unload the result. The simplicity of this approach stems from the fact

that the algorithms to LOAD and UNLOAD the array processor have already been developed. The flexibility comes from being able to omit LOADing or UNLOADing operations if the matrices are already in the array, or if the result is to remain in the array. The actual ADDition operation is achieved by instructing each PE to add the two elements which it either received or had from previous operations.

Since LOAD, ADD and UNLOAD have the attributes of wavefront algorithms, their execution can be overlapped to reduce the total time needed to add two matrices. The starting set-up for matrix addition is shown in figure 3.12, along with the summary of the calculation carried out by each PE as given in figure 3.13. The SUBtraction algorithm is summarized in the same figure.

3.3.2 Addition of Three or More Matrices

Repeated ADDition instructions allow more than two matrices to be added without any need to unload intermediate sums. Figure 3.14 illustrates the set-up for repeated additions. Notice how the matrices to be added are flowed into the array one after another with only an ADD instruction between them. This technique can be extended indefinitely to add any number of matrices.

3.3.3 Scaling Matrices by a Constant

The scaling of a matrix by a constant factor is accomplished on the array processor in a similar fashion to matrix addition and subtraction. The matrix is LOADed, then the SCALE instruction and the scaling factor are passed through

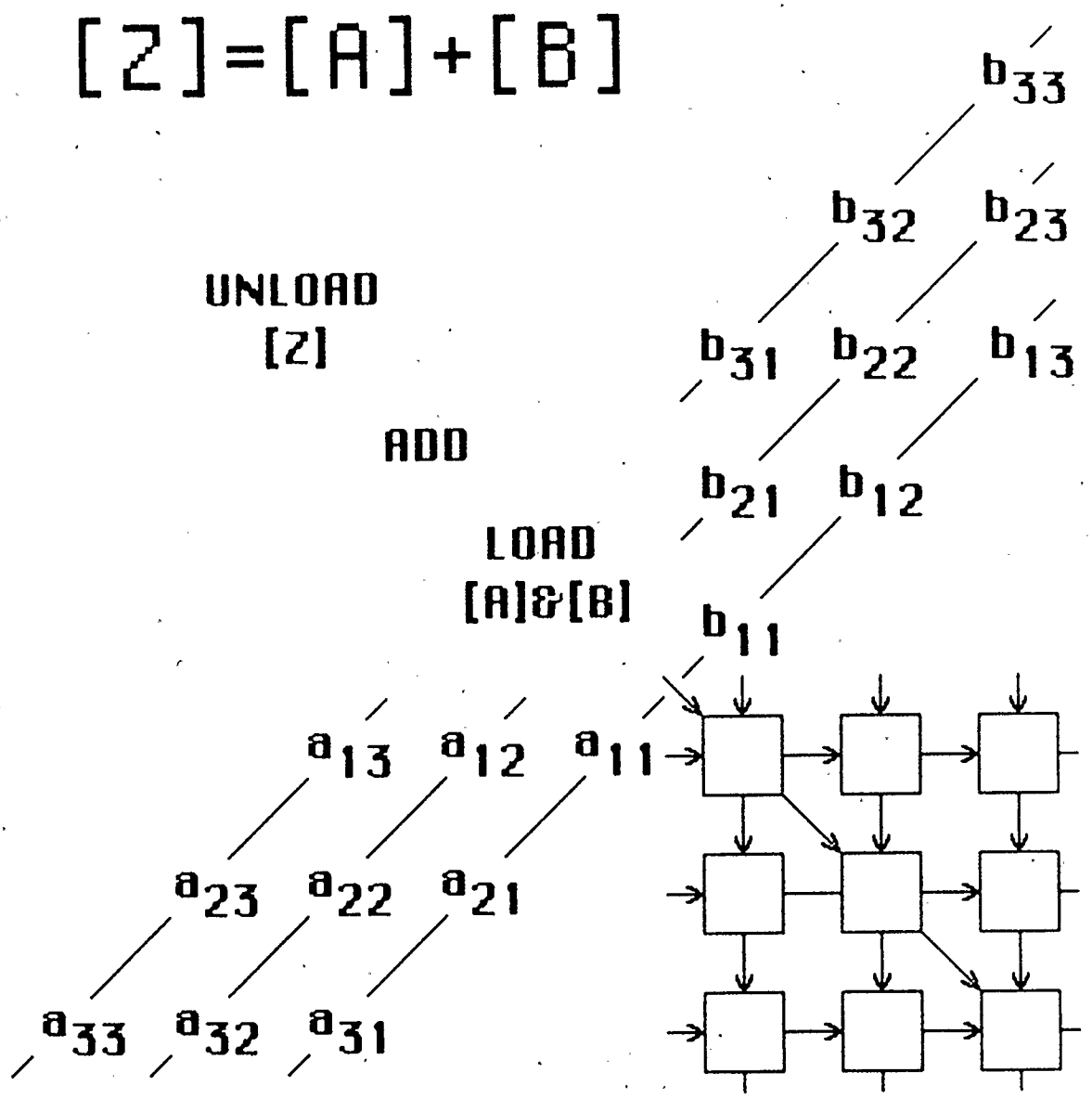


Fig. 3.12: Set up for matrix addition using LOAD, ADD and UNLOAD.

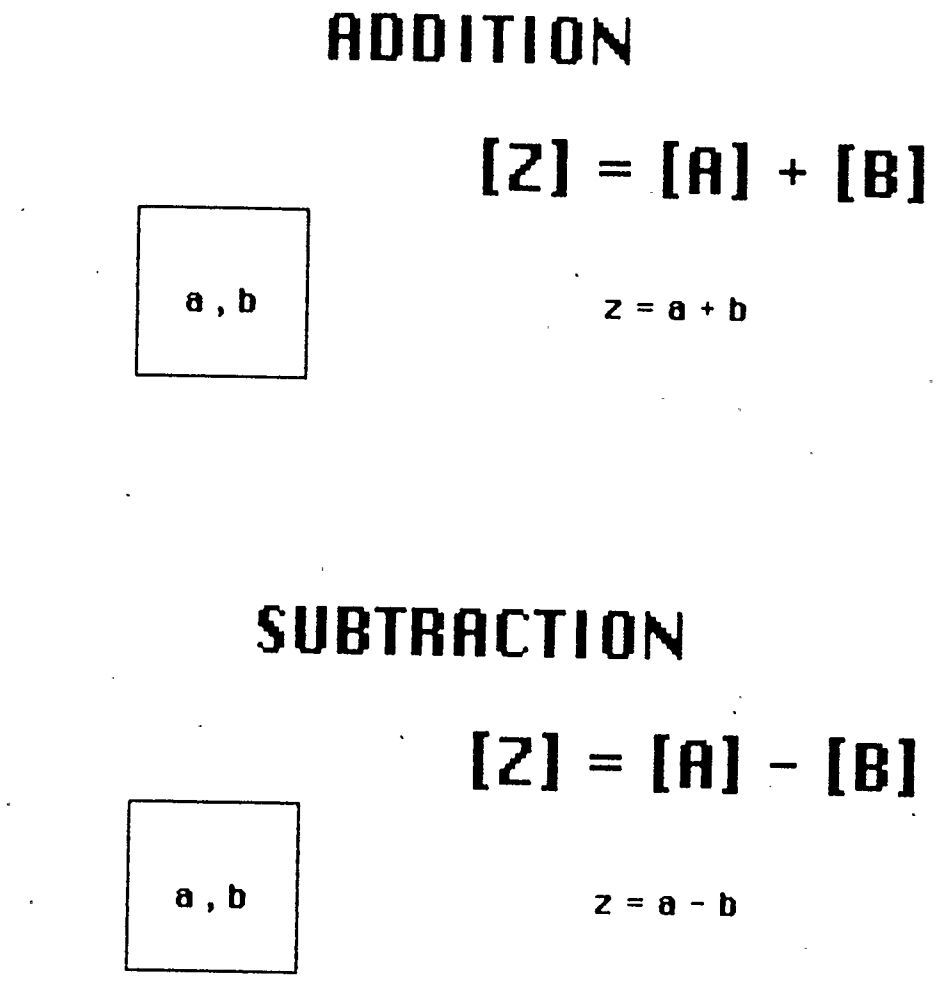


Fig. 3.13: Summary of the matrix ADDition and SUBtraction algorithms.

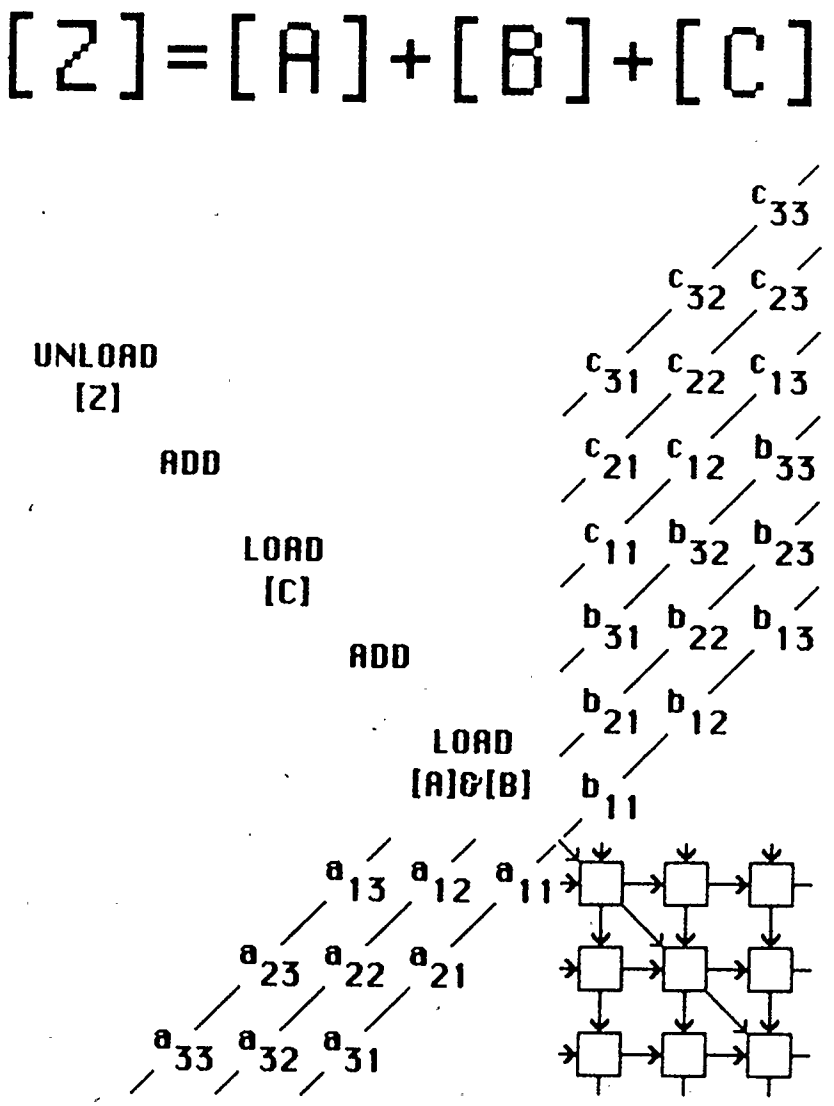


Fig. 3.14: Adding three matrices by repeated ADD operations.

the array. Each PE multiplies its matrix element by the constant. This simple operation is summarized in figure 3.15.

The set-up for the scaling operation is shown in figure 3.16. This figure assumes that the array already contains the matrix. The result of the scaling operation remains in the array unless specifically UNLOADed.

3.3.4 Addition of Large Matrices by Partitioning

The addition of matrices with dimensions exceeding the size of array requires partitioning the matrices into sub-matrices. The corresponding sub-matrices are added together to give the sub-matrices which make up the sum.

The set-up for partitioned matrix addition is shown in figure 3.17. It is similar to the set-up for repeated matrix additions except that the sub-matrices are UNLOADed between ADD operations.

When matrix addition is performed using partitioned matrices, there is no extra I/O requirement imposed on the system. Each matrix element is needed only once. There is, however, a penality in speed when compared to the amount of time needed to add the same matrices using a larger array. For square matrices that are k times larger than the array, the time needed to evaluate the addition is approximately

$$\frac{(k^2 + 1)}{k} \qquad (3.1)$$

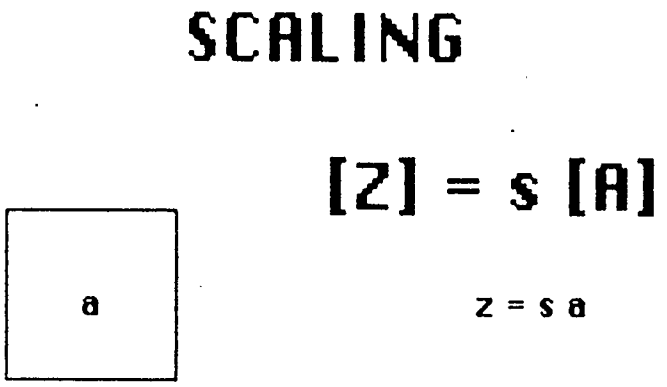


Fig. 3.15: Summary of the SCALE algorithm.

$$[Z] = s[A]$$

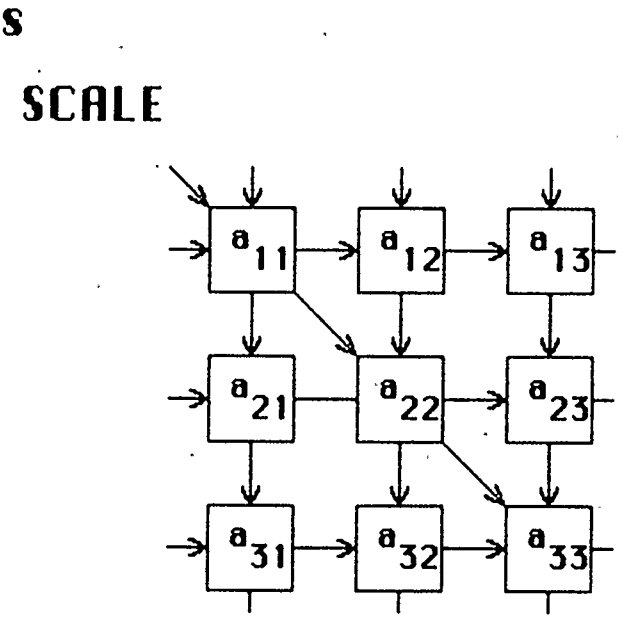


Fig. 3.16: Scaling a matrix by a constant.

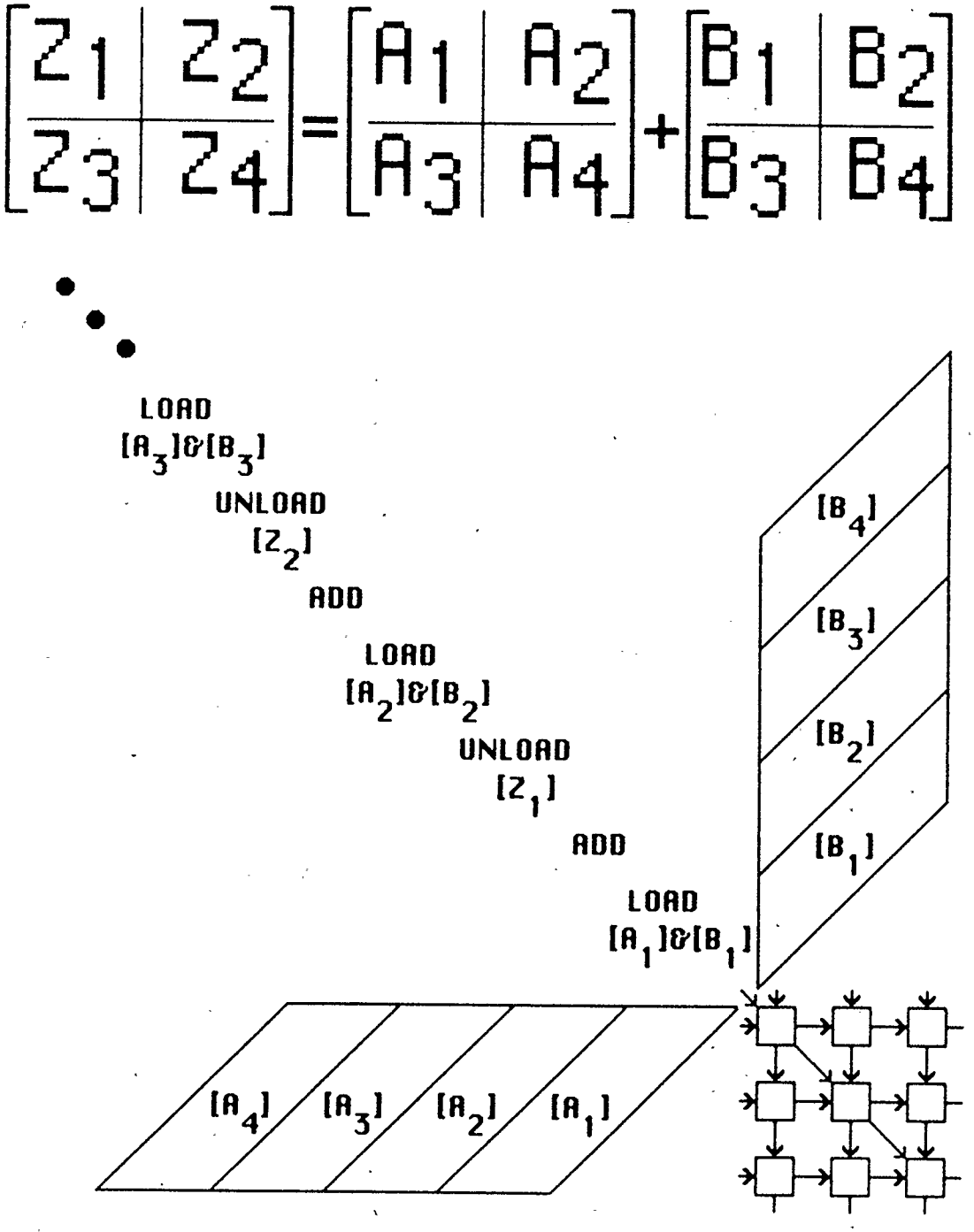


Fig. 3.17: Set-up for partitioned matrix addition.

times as long as would be needed if the array was as large as the matrices. Table 3.1 shows the increase in computational time for several values of k.

k	Time Increase Factor
1	1.0
2	1.25
3	1.67
4	2.125
5	2.6

Table 3.1: Computation time for partitioned matrix addition.

3.4 MULTIPLYING MATRICES

Matrix multiplication is a computationally bound operation. A parallel processing system can substantially reduce the time needed to multiply matrices. Considering the large number of applications that require the ability to multiply matrices, it is not surprising that so much research has gone into designing systolic matrix multiplication algorithms. This research is reviewed in the next section.

3.4.1 Previous Work on Matrix Multiplication Algorithms

Multiplication of a matrix by a vector using a systolic architecture was considered by H.T. Kung and C.E. Lieserson [2] even before the term "systolic" appeared in the literature. Matrix-vector multiplication is a specific case of the more general matrix-matrix multiplication operation. The same article describes a matrix-matrix multiplication algorithm which flows two matrices into a systolic array from different directions and flows out the product of these matrices in a third direction.

This algorithm is interesting for a number of reasons: firstly, it reduces the time complexity of matrix multiplication from $O(n^3)$ to $O(n)$; secondly, it uses simple regularly connected PEs that all execute the same sequence of operations; and finally, band matrices can be multiplied on an array of processors whose dimensions are equal to the widths of the bands of the matrices. The layout of this design is shown in figure 3.18.

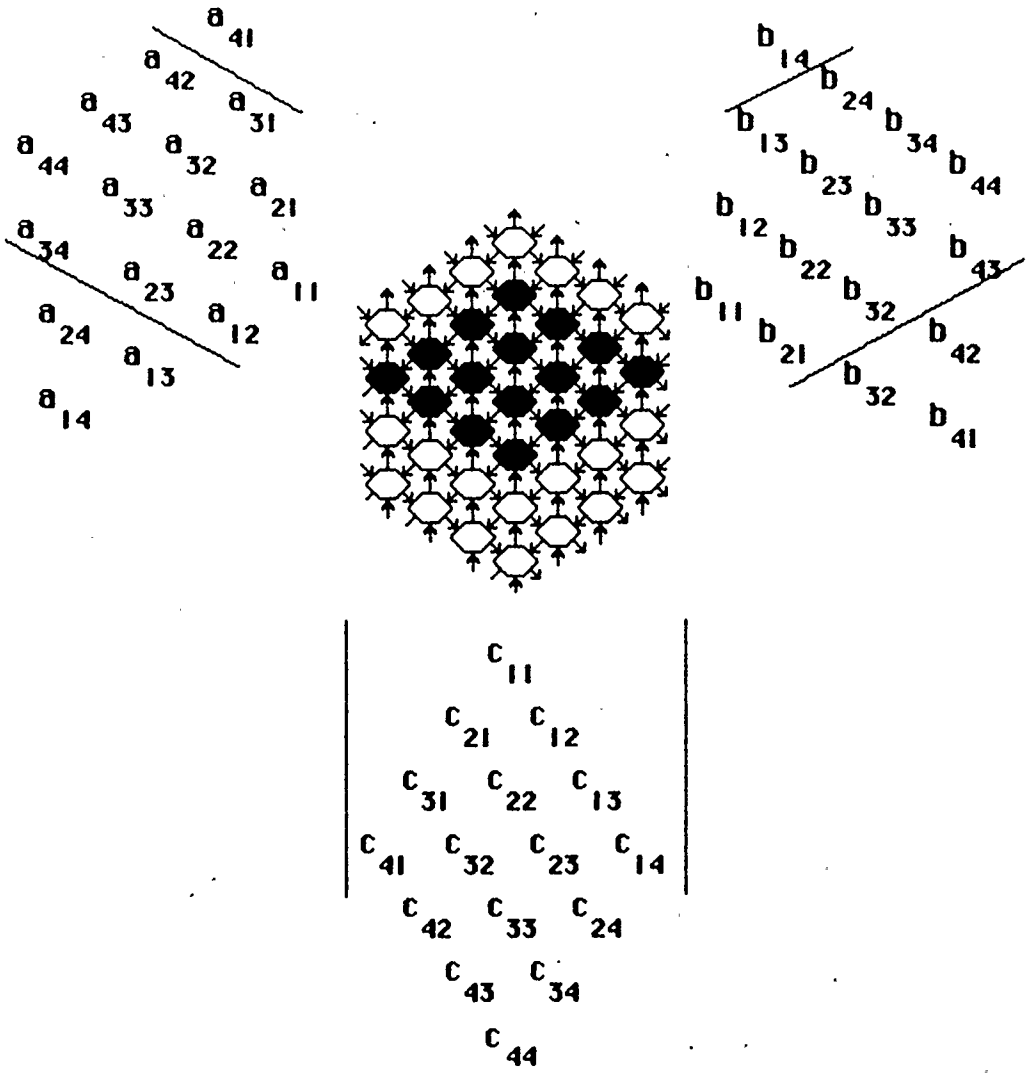


Fig. 3.18: Hexagonally connected systolic array for matrix multiplication.

There are, however, several drawbacks to this design. Firstly, the input format brings the matrices into the array in a direction parallel to the main diagonal of the matrix. If the matrices are densely populated, the size of the array needed to accommodate the matrices has dimensions that are one less than the sum of the dimensions of each matrix.

Secondly, the array of hexagonally connected processors is in the shape of a hexagon. This is an awkward shape to use efficiently for matrix operations other than matrix multiplication. The article by Kung and Lieserson fails to reveal this problem because the dimensions of the band matrices used in their example lead to a special case where the shape of the array becomes square. The dotted lines and the shaded PEs in figure 3.18 indicate the matrices used in their example.

Thirdly, only one of every three processors is active during each step of the algorithm. This suggests that an algorithm could be developed which uses one third of the number of processors required in this algorithm. Lastly, as there is no opportunity to leave the product matrix in the array, it must flow out of the array for the algorithm to work. All of these difficulties tend to make this design suitable only for the dedicated task of multiplying matrices. A general purpose systolic array processor should not have these properties.

Priester et al [12] describe alternative arrangements for inputting matrices that reduce the required array size down to either the number of rows or

the number of columns in the input matrices.

S.Y. Kung et al [11] make use of an input format similar to one of those described by Priester. This is the same format used by the LOAD and UNLOAD algorithms described earlier. Besides reducing the size of the array, this format changes the shape of the array from hexagonal to rectangular and keeps each processor active all of the time. Each processor performs the same sequence of steps as in H.T. Kung's design. Consequently, fewer processors are needed to perform the task of matrix multiplication. The product of the two matrices multiplied on an array of this design resides in the array at the end of the multiplication operation. A method of unloading results from the array is required, but being able to leave the product in the array increases the usefulness of the algorithm. This design is shown in figure 3.19 and the algorithm is summarized in figure 3.20. Kung's algorithm is abbreviated to "MULT 2" for use in figures.

A technique described by Hoare [13] for multiplying matrices uses the same rectangular array used by S.Y. Kung and the same input format, but begins with one matrix already loaded into the array. The second matrix flows into the array and the product of the two matrices flows out of the array.

Figure 3.21 shows the layout for this technique. Notice that the matrix that enters the array from the north is transposed compared to the arrangement used in S.Y. Kung's algorithm. Also notice that if the initial values that flow into the array from the east are the elements of another matrix instead of zeros, then

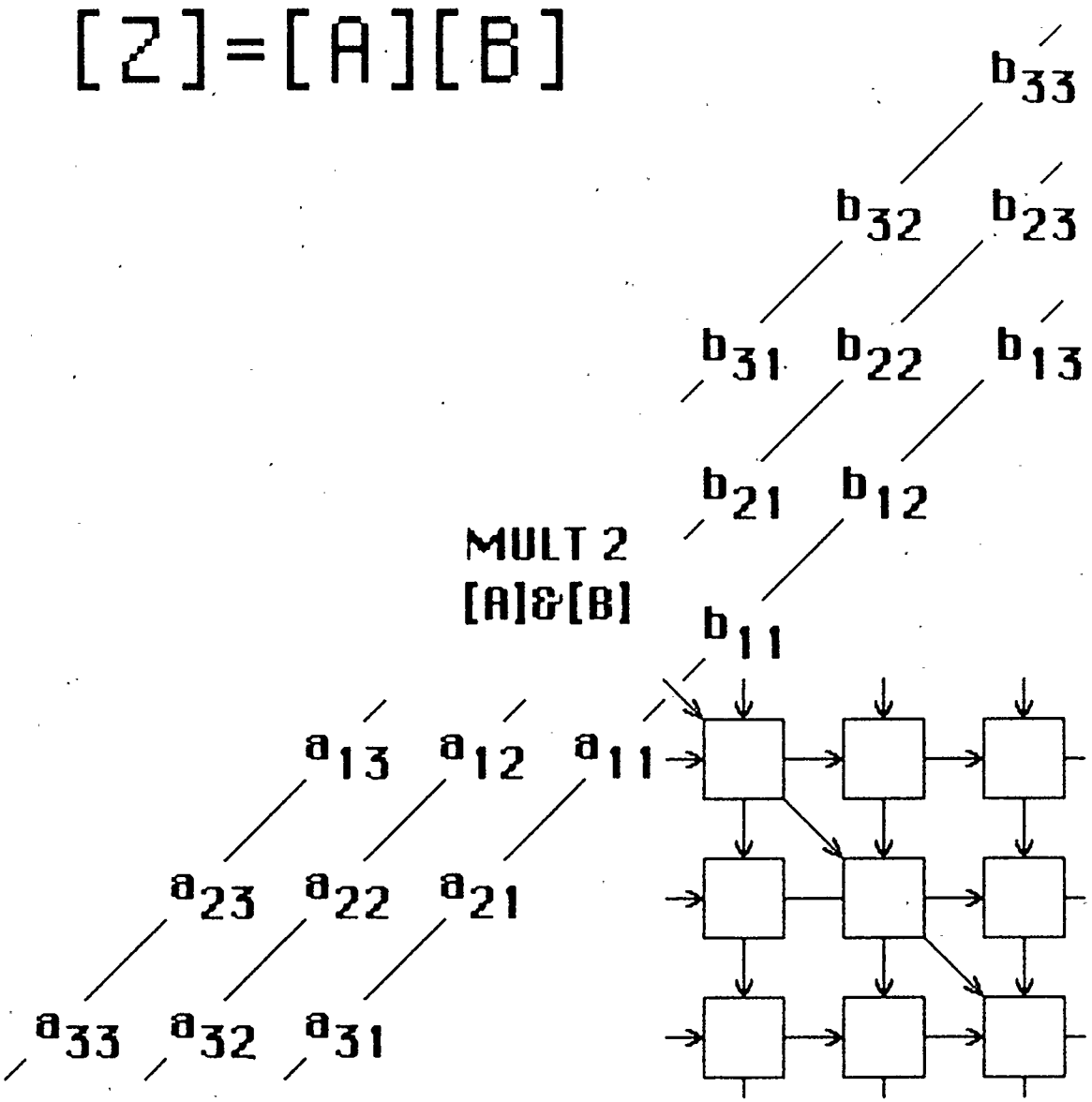
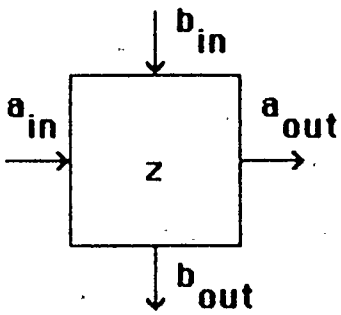


Fig. 3.19: A square array for multiplying matrices.

MULTIPLICATION 2

[Z] = [A][B]



$z = 0$

for all other pairs a_{in}, b_{in} :

$a_{out} = a_{in}$

$b_{out} = b_{in}$

$z = z + a_{in} b_{in}$

Fig. 3.20: Summary of S.Y Kung's matrix multiplication algorithm.

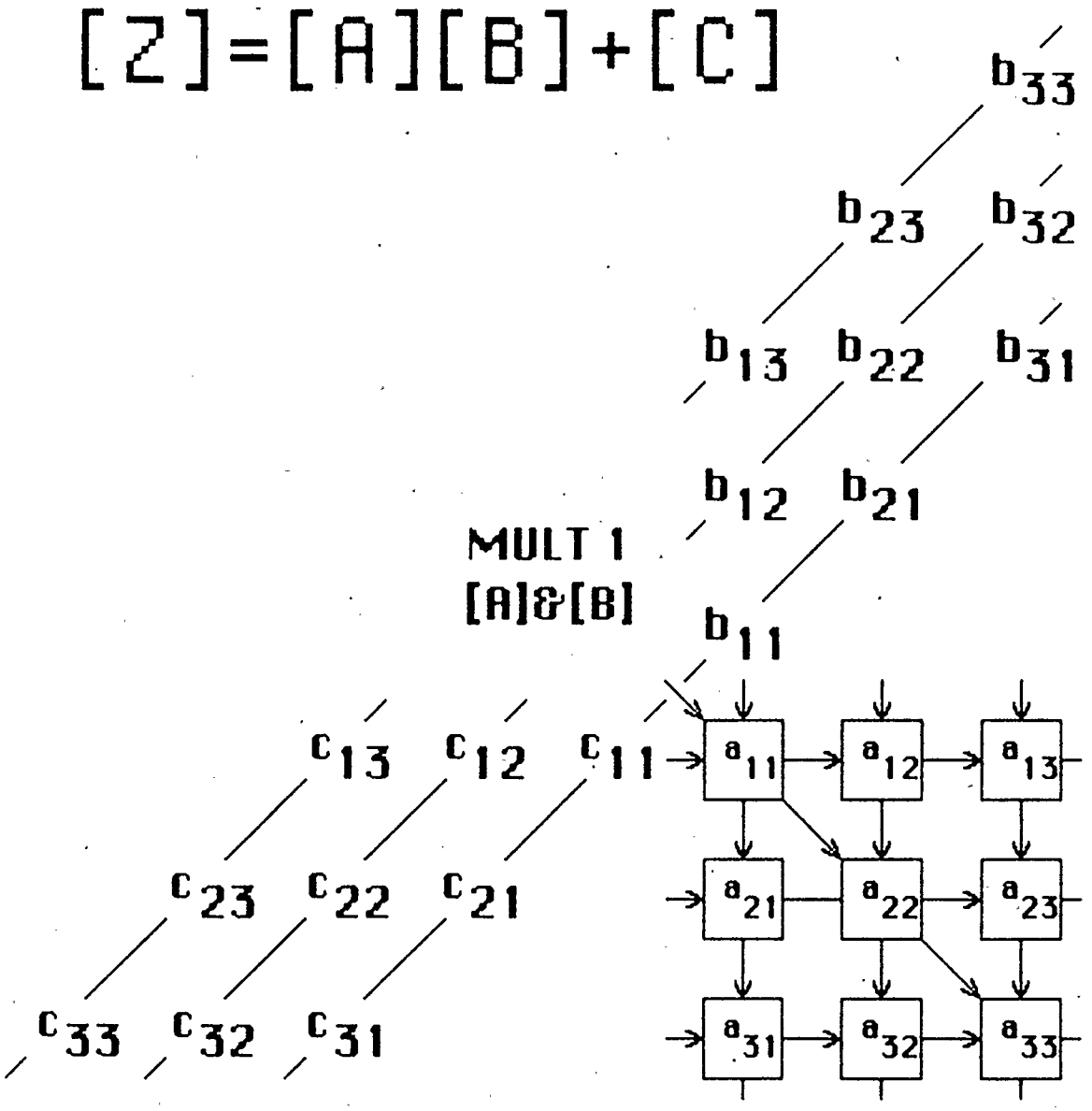


Fig. 3.21: Arrangement of data for Hoare's matrix multiplication algorithm.

Hoare's algorithm calculates an equation of the form:

$$[Z] = [A][B] + [C] \tag{3.2}$$

Figure 3.22 summarizes the calculations performed during each step of this algorithm. Hoare's matrix multiplication algorithm is referred to as "MULT 1" in the figures.

Hoare's matrix multiplication algorithm post-multiplies the matrix in the array by the matrix that is flowed in. Flowing the input matrix into the array from the west rather than the north results in the pre-multiplication operation. This variation of Hoare's algorithm is shown in figure 3.23. Notice that the initial values of the product matrix are flowed in from the north and that the product flows out from the south.

The next two sections consider how MULT 1, MULT 2, LOAD and UNLOAD can be used to multiply matrices in a number of different circumstances.

3.4.2 Multiplying Two Matrices

Combining S.Y. Kung's algorithm with the UNLOAD operation yields a procedure which duplicates the function of H.T. Kung's algorithm while avoiding all of the disadvantages mentioned earlier. The execution of this combination of algorithms can be overlapped since both the UNLOAD operation and S.Y. Kung's multiplication algorithm exhibit wavefront properties. The sequence of multiplying the two matrices followed by unloading the product matrix from the array takes the

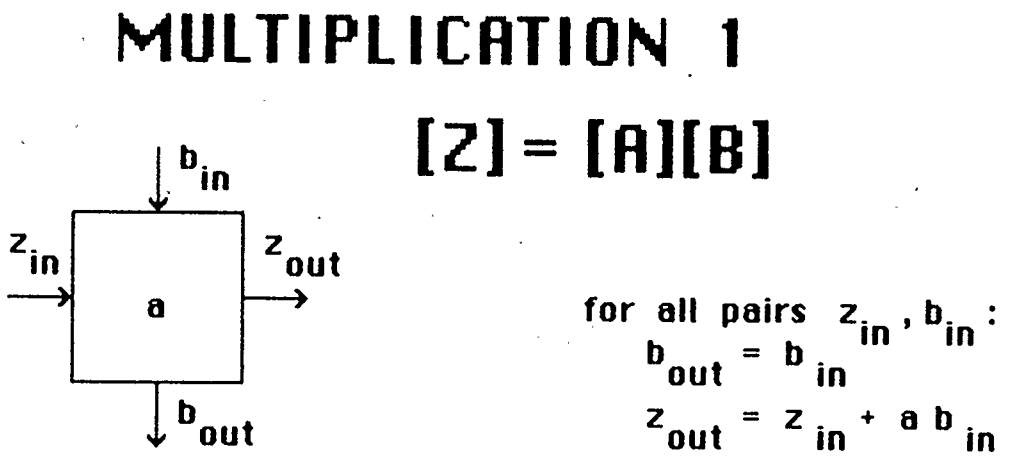


Fig. 3.22: Summary of MULT 1 algorithm of Hoare.

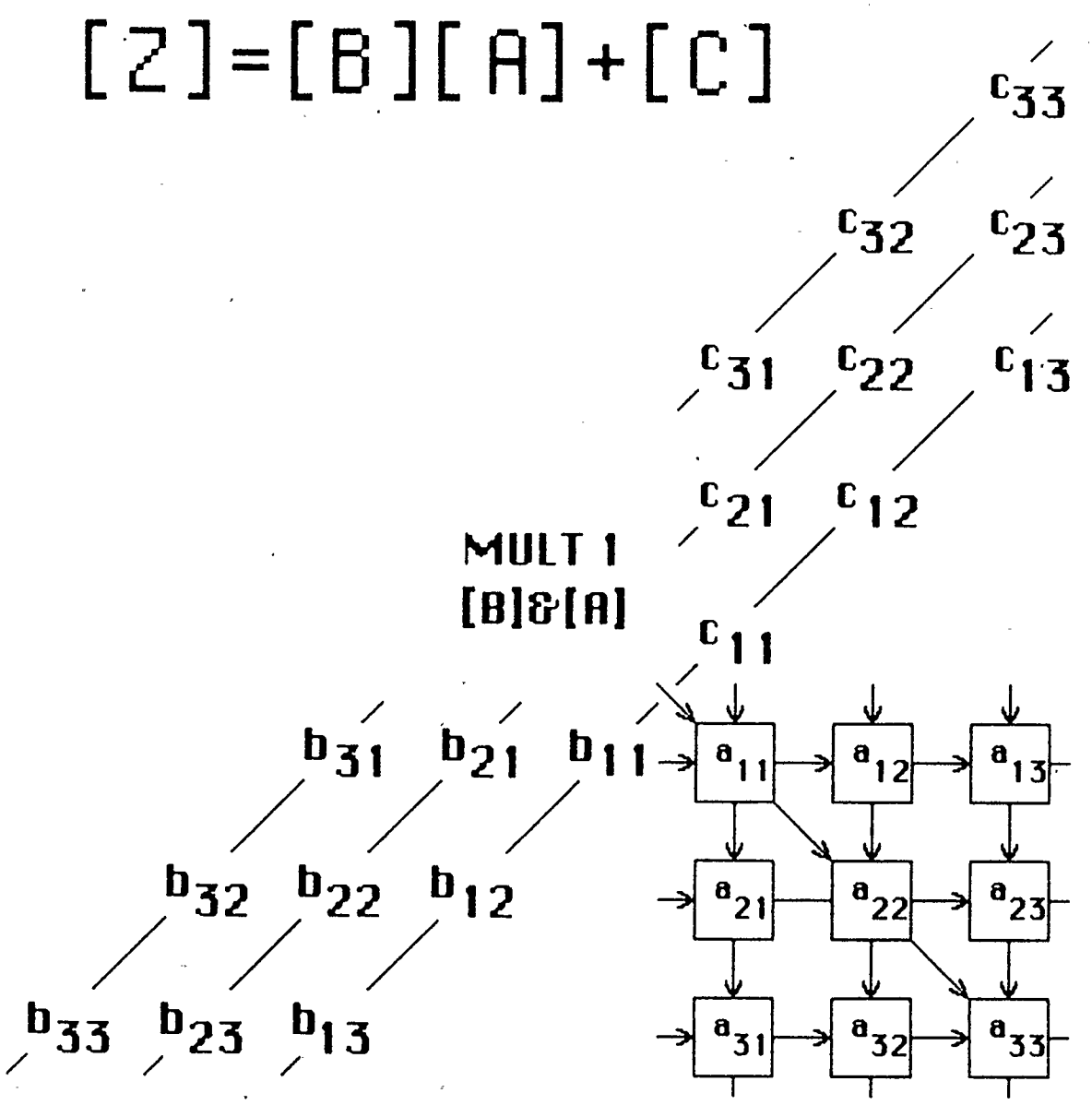


Fig. 3.23: The arrangement used to preMULTiply using Hoare's algorithm.

same number of steps as H.T. Kung's algorithm.

Alternatively, the LOAD procedure followed by the algorithm of Hoare could be used to duplicate the function of multiplying two matrices with the product flowing out of the array. The same number of steps are required as in H.T. Kung's algorithm and as in S.Y. Kung's algorithm followed by the UNLOAD operation.

In the course of evaluating matrix equations, multiplication of a previous intermediate result by another matrix is commonly required, as is the need to perform some other operation on the product of two matrices. In these cases, Hoare's algorithm and S.Y. Kung's algorithm, respectively, satisfy the requirements without using LOAD or UNLOAD.

3.4.3 Multiplying Three Matrices

The algorithm of S.Y. Kung flows in two matrices, A and B, and leaves their product, matrix Y, in the array. The algorithm of Hoare flows in a matrix C and multiplies it by the matrix already in the array causing the product matrix Z to flow out of the array. Combining these two algorithms, as shown in figure 3.24, allows three matrices to be multiplied together without the intermediate result Y leaving the array.

Both of the algorithms have wavefront properties, therefore, they can be overlapped to reduce the number of steps required to multiply three matrices.

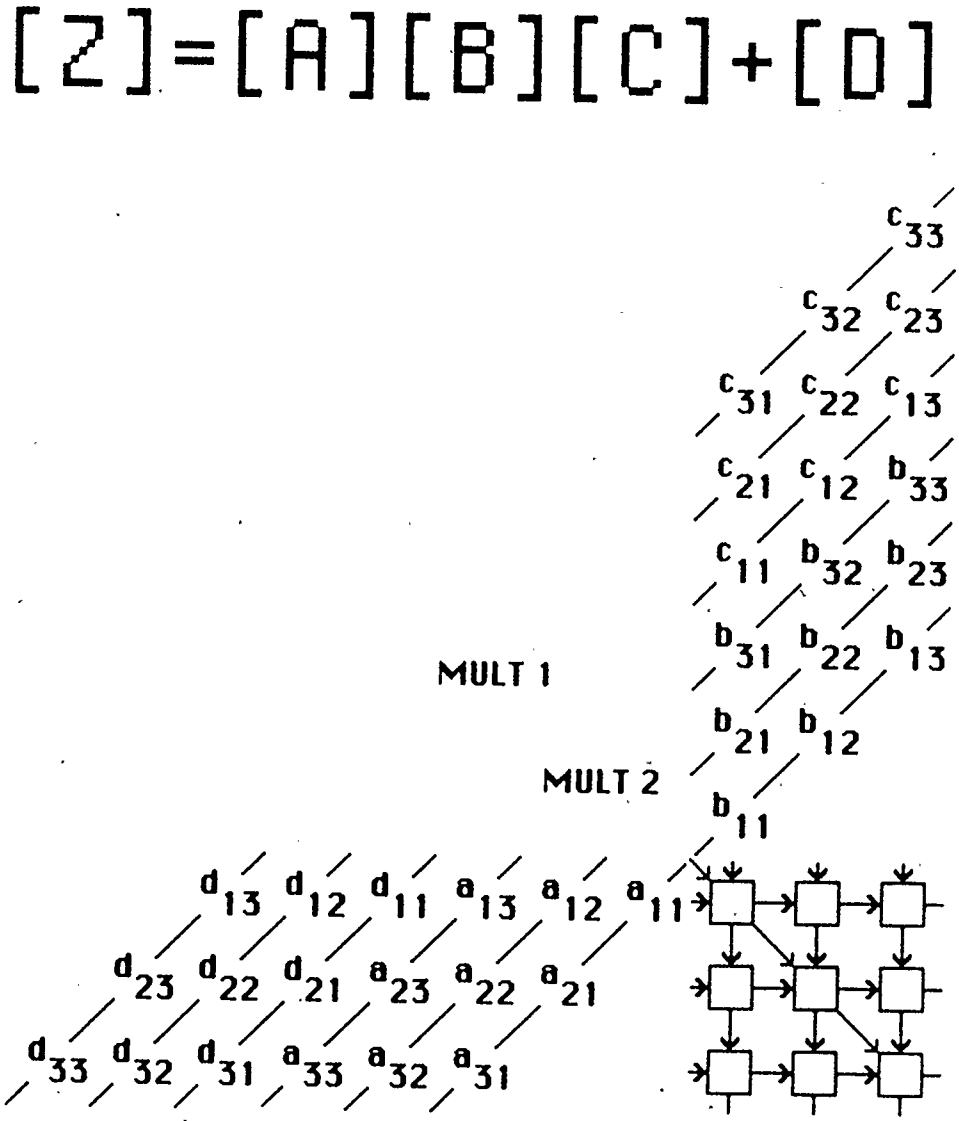


Fig. 3.24: Multiplying three matrices using MULT 1 and MULT 2.

The flexibility in being able to efficiently combine algorithms to accomplish tasks reduces the number of algorithms needed to cover the possible situations where matrix multiplication is involved in evaluating matrix equations.

3.4.4 Multiplying Four Matrices and Beyond

Unfortunately, multiplying four or more matrices in succession cannot be done using combinations of the algorithms described above without UNLOAD-ing and reLOADing intermediate products. The existence of an algorithm that flows in one matrix, multiplies it by a matrix already in the array, and leaves the product in the array would allow multiplying any number of matrices efficiently. Such an algorithm has eluded discovery. However, while products of two or three matrices are quite common in matrix equations, products of four or more matrices are not. Rearranging the form of equations might allow instances of such products to be eliminated altogether.

3.4.5 Multiplying Large Matrices Using Partitioning

The use of partitioning to multiply matrices with dimensions exceeding the size of the array is similar to matrix multiplication on a single processor. Instead of multiplying corresponding rows and columns of elements, the array multiplies rows and columns made up of sub-matrices.

If a pair of matrices are partitioned into sub-matrices that are half the size of the original matrix as shown below,

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} Z_1 & Z_2 \\ Z_3 & Z_4 \end{bmatrix} \tag{3.3}$$

then the sub-matrices of the product matrix Z are given by:

$$[A_1][B_1] + [A_2][B_3] = [Z_1] \tag{3.4}$$

$$[A_1][B_2] + [A_2][B_4] = [Z_2] \tag{3.5}$$

$$[A_3][B_1] + [A_4][B_3] = [Z_3] \tag{3.6}$$

$$[A_3][B_2] + [A_4][B_4] = [Z_4] \tag{3.7}$$

To evaluate these equations on the array processor requires eight sub-matrix multiplications and four additions. Each sub-matrix is used twice, so the amount of I/O is doubled. The entire process requires slightly more than the expected factor of eight increase because of the extra overhead associated with passing the instructions through the array. The set-up for this partitioning example is shown in figure 3.25.

$$\begin{bmatrix} Z_1 & Z_2 \\ Z_3 & Z_4 \end{bmatrix} = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix}$$

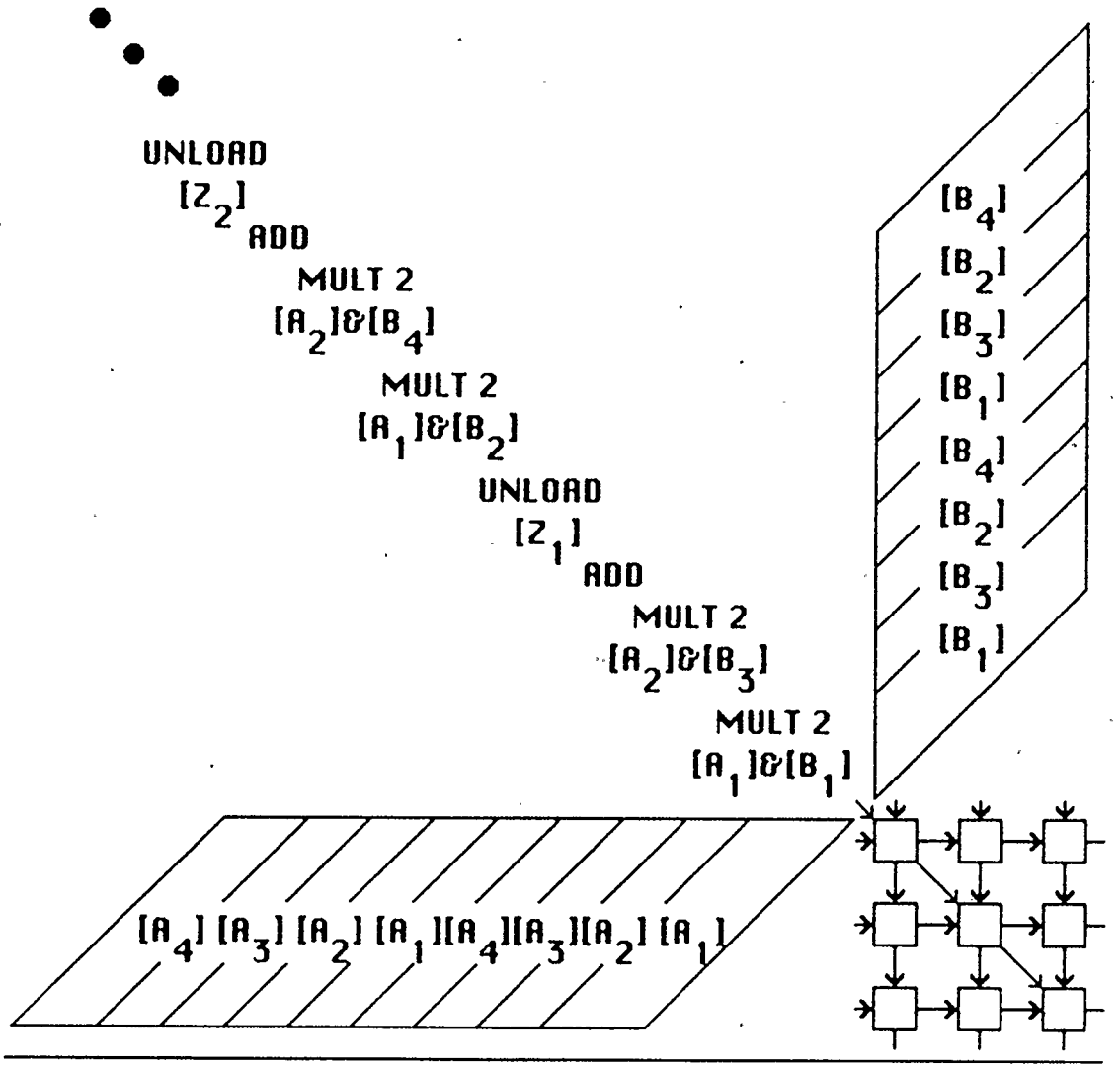


Fig. 3.25: Arrangement of sub-matrices for partitioned multiplication.

3.5 MATRIX INVERSION

Solution of a system of equations expressed using the matrix form:

$$[A][x] = [b] \tag{3.8}$$

is a commonly encountered problem in engineering and science. The calculation of the inverse of a matrix A is the solution of the above equation with vector b set equal to each column of the identity matrix I. The resulting solution vectors comprise the columns of the inverse matrix. The inverse of a matrix of order n requires n times as much work as solving a system of n equations. This leads to the conclusion [14] that solving equations by first calculating the inverse matrix requires many times more effort than solving the equations directly.

It should also be noted that although matrix equations often indicate the need to invert a matrix, usually the inverted matrix is immediately multiplied by a vector and therefore the matrix and the vector should be solved rather than inverting the matrix. The occurrence of inverses in matrix equations is frequently just for mathematical convenience so that often much effort can be saved by evaluating the equation as two separate equations. For example:

$$\alpha = [y]^t[A]^{-1}[b] \tag{3.9}$$

should be evaluated by first solving:

$$[A][x] = [b] \tag{3.10}$$

and then calculating alpha using:

$$\alpha = [y]^t[x] \tag{3.11}$$

The methods described in the next sections can be applied to both solving systems of equations and to inverting matrices.

3.5.1 The Cofactor Method

The inverse of a square matrix A is that matrix which when multiplied by matrix A gives the identity matrix I. From this method comes the formula:

$$[Z] = [A]^{-1} = \frac{[cof\ A]^t}{\det A} \tag{3.12}$$

The cofactor method is commonly used to invert matrices of order two or three by hand. However, for higher order matrices the calculation of the cofactors and the determinant becomes a computational nightmare. To understand why this problem occurs, consider how the determinant is calculated. The determinant is the algebraic sum of the products of all permutations of the matrix elements taken in groups of n such that one and only one element is taken from each row and column of the matrix. Even permutations are given a positive sign, odd permutations have a negative sign. The number of permutations that must be evaluated is given by:

$$(n)(n-1)(n-2)\ \cdots\ (2)(1) = n! \tag{3.13}$$

where the symbol "!" denotes the factorial function.

Each of the n! permutations involves $(n-1)$ multiplications, giving a total of $(n!)(n-1)$ multiplications to find the determinant of order n, together with $n!-1$ additions. Computing the cofactor of an element is very similar to computing

the determinant of a matrix except that none of the elements in the same row or column as the element whose cofactor is being found are included in the calculations. In other words, the cofactor of element (i,j) is the determinant of the $(n-1)th$ order matrix formed by eliminating row i and column j from the original matrix.

Table 3.2 gives the number of multiplications and additions needed to calculate the determinant of matrices of order two through ten.

Grouping similar terms when evaluating the determinant reduces the number of multiplications and additions only by a small factor. The factorial nature of the cofactor method overwhelms any effort to reduce the computational complexity. A parallel processing system would have only limited success in reducing the time required to compute the inverse by this method. Parallelism does exist in the cofactor and determinant calculations but it is not nearly sufficient to account for the factorial computational complexity.

3.5.2 Gaussian Elimination

The technique of Gaussian elimination (GE) uses the principle which states that if identical row and column operations are performed on a matrix A and on the identity matrix I such that matrix A is transformed into the identity matrix, then the identity matrix will be transformed into the inverse of matrix A.

The set of elementary row and column operations [15] that are used to eliminate elements off the main diagonal of the matrix A are:

Order	# of terms = n!	# of mult = n!(n-1)	# of additions = n!-1
2	2	2	1
3	6	12	5
4	24	72	23
5	120	480	119
6	720	3600	719
7	5040	30240	5039
8	40320	282240	40319
9	362880	2901040	362879
10	3628800	32659200	3628799

Table 3.2: Computational complexity of the cofactor method.

-multiply a row by a non-zero constant,

-interchange two rows,

-add a multiple of one row to another row.

These operations apply to columns also.

The order of elimination of off diagonal elements is shown in figure 3.26. All the elements in the column extending above and below the element on the main diagonal are eliminated before the elements in the next column. The element on the main diagonal is called the pivot. Since calculation of the multiple of the pivot row to be added to the other rows to eliminate off diagonal elements involves dividing by the value of the pivot element, it is desirable to keep the pivot values as large as possible. Techniques for selecting pivot elements are discussed in the section on pivoting.

The entire procedure for GE including pivoting is summarized by the following steps:

1) Select the most appropriate element to be the pivot.

2) Move the pivot element onto the main diagonal by performing row and column interchanges.

3) Use the pivot value to calculate the multiplies of the pivot row needed to eliminate the off diagonal elements in the same column as the pivot.

4) Repeat steps 1), 2) and 3) for each column in the matrix, and

p1	2	3	4	5
1	p2	3	4	5
1	2	p3	4	5
1	2	3	p4	5
1	2	3	4	p5

Fig. 3.26: Order of elimination of off diagonal elements.

duplicate the same operations on the identity matrix.

3.5.3 Pivoting Techniques

Interchanging rows or columns such that the value of the pivot element does not equal zero is called pivoting. It is also desirable to employ pivoting when the pivot value is small in order to reduce the effects of round-off error on the inverse matrix values. Round-off errors are inevitable when a finite word length is used to represent numbers.

All pivoting strategies seek to reduce round-off errors by selecting the largest element from a portion of the matrix to be the pivot value. Three of the commonly used pivoting techniques are listed below in order of increasing difficulty and increasing success in reducing round-off errors [16]:

1) partial pivoting (maximal column pivoting),

2) scaled column pivoting,

3) total pivoting (maximal pivoting).

Partial pivoting chooses the pivot to be the element in the pivot column with the largest absolute value. Scaled column pivoting first normalizes each row of the matrix by dividing the elements in each row by the row's largest element, then proceeds to choose the element in the pivot column with the largest absolute value to be the pivot. Total pivoting uses the element with the largest

absolute value to be the pivot. Note that, regardless of which pivoting strategy is used, only one element from each row and column can be used as a pivot. Thus, the number of elements that need to be considered for each successive pivot position decreases.

3.5.4 Gaussian Elimination on a Systolic Array

The following discussion illustrates how GE is performed under the constraints imposed by the wavefront architecture.

This GE algorithm differs markedly from the other wavefront algorithms because it allows the source of the wavefronts to move within the array. The GE procedure is carried out on a matrix already in the array.

Figure 3.27 shows the different operations that are carried out to eliminate the off-diagonal elements in column two. PE(2,2) is the source and its matrix element is the pivot. PEs in the same row as the pivot divide their matrix element by the pivot value and pass the result to the north and south. PEs in the same column as the pivot pass their matrix element to the east and west before replacing its value with zero. All other PEs update their matrix element by subtracting the product of the ratio passed along their column and the factor passed along their row. The designation of "source" is then passed on to the next diagonal PE and the procedure is repeated. The same operations are performed on the identity matrix. Each step of GE takes a fixed number of steps, so that the algorithm has a

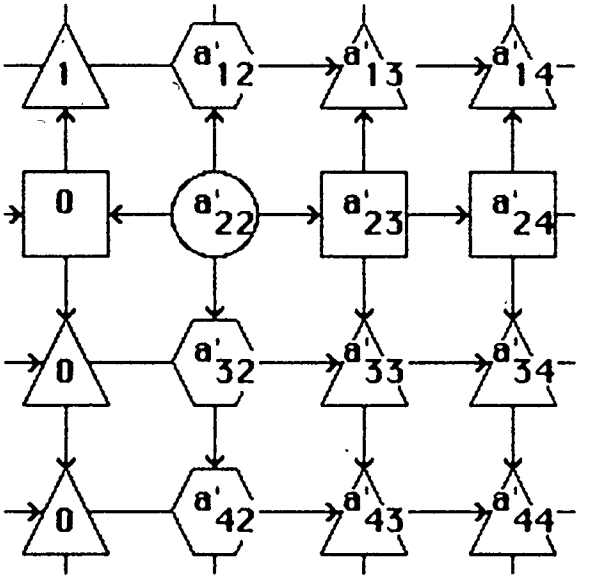


Fig. 3.27: A wavefront array executing a step of GE.

linear time complexity.

Because GE uses a moving source of wavefronts, extra steps are required at the end of the GE process to move the source back to PE(1,1). All instructions begin with PE(1,1) as source, so all instructions must end this way.

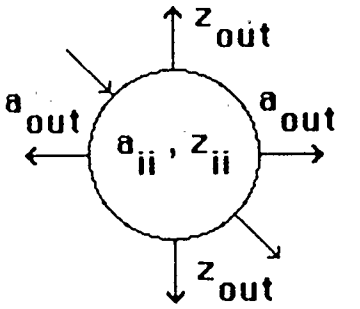
Figure 3.28 gives the calculations performed by each type of PE involved in the GE procedure. The next section considers how pivoting is implemented.

3.5.5 Pivoting Methods for Gaussian Elimination

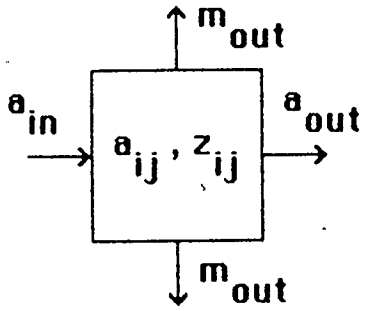
Partial pivoting and total pivoting are both desirable additions to the GE algorithm.

Partial pivoting requires a magnitude comparison to be done for all the elements below and including the pivot position element to find the element with the largest magnitude. Row interchanges are used to move the pivot onto the main diagonal. Notice that the rows of the identity matrix must also be interchanged to correspond to the changes made to the original matrix. Unfortunately, the comparison process and the interchange process tend to be sequential and involve only a few of the PEs. None of the other PEs can begin the next step of GE until the pivot is selected. The limitation of passing only one piece of data per step makes partial pivoting cumbersome to implement.

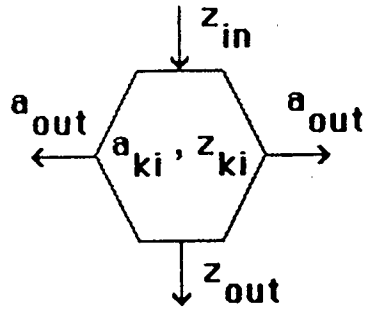
GAUSSIAN ELIMINATION



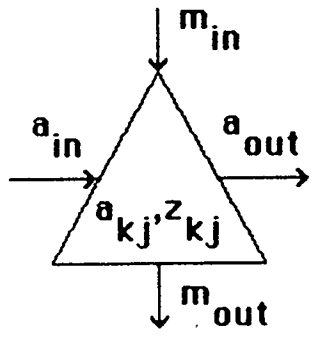
$$a_{out} = a_{ii}$$
$$z_{ii} = z_{ii} / a_{ii}$$
$$z_{out} = z_{ii}$$



$$a_{ij} = a_{ij} / a_{in}$$
$$z_{ij} = z_{ij} / a_{in}$$
$$a_{out} = a_{in}$$
$$m_{out} = a_{ij}$$
$$m_{out} = z_{ij}$$



$$z_{ki} = z_{ki} - z_{in}\, a_{ki}$$
$$z_{out} = z_{in}$$
$$a_{out} = a_{ki}$$
$$a_{ki} = 0$$



$$a_{kj} = a_{kj} - m_{in}\, a_{in}$$
$$z_{kj} = z_{kj} - m_{in}\, a_{in}$$
$$a_{out} = a_{in}$$
$$m_{out} = m_{in}$$

Fig. 3.28: Summary of the GE algorithm.

Figure 3.29 summarizes the partial pivoting procedure. Figure 3.30 shows the steps involved in the partial pivoting procedure. The rows of the matrices are not interchanged a row at a time as suggested by the figure. The wavefront nature of the pivoting algorithm results in adjacent PEs interchanging their matrix elements with the row of PEs above them during successive steps.

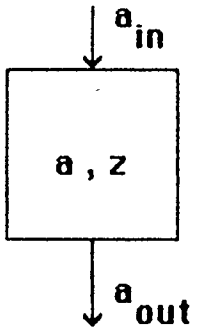
Total pivoting requires that all of the matrix elements in a certain sub-matrix be compared to find the element with the largest magnitude. This sub-matrix is the original matrix with all rows and columns containing a previously chosen pivot deleted. Row interchanges and column interchanges are used to move the selected pivot to the pivot position.

Figure 3.31 shows the procedure for total pivoting. Figure 3.32 shows the steps involved in total pivoting with a(3,4) chosen to be the pivot. The PEs on the main diagonal must find the largest of four values instead of only two, however, they have twice as long as the off-diagonal PEs to do the comparisons. Some of the PEs perform their comparison step concurrently, thus, the total pivoting algorithm requires the same number of comparison steps as the partial pivoting algorithm.

3.5.6 LU Decomposition Methods

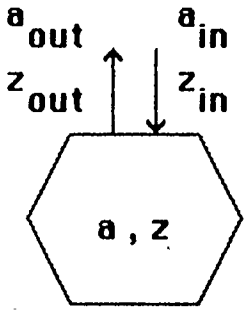
Another approach to the problem of inverting matrices and solving systems of equations is to decompose the original matrix into the product of two or

PARTIAL PIVOTING



Compare Magnitudes:

$a_{out} = \max(|a|, |a_{in}|)$



Interchange Rows:

$a_{out} = a$

$z_{out} = z$

$a = a_{in}$

$z = z_{in}$

Fig. 3.29: Summary of the Partial Pivoting algorithm.

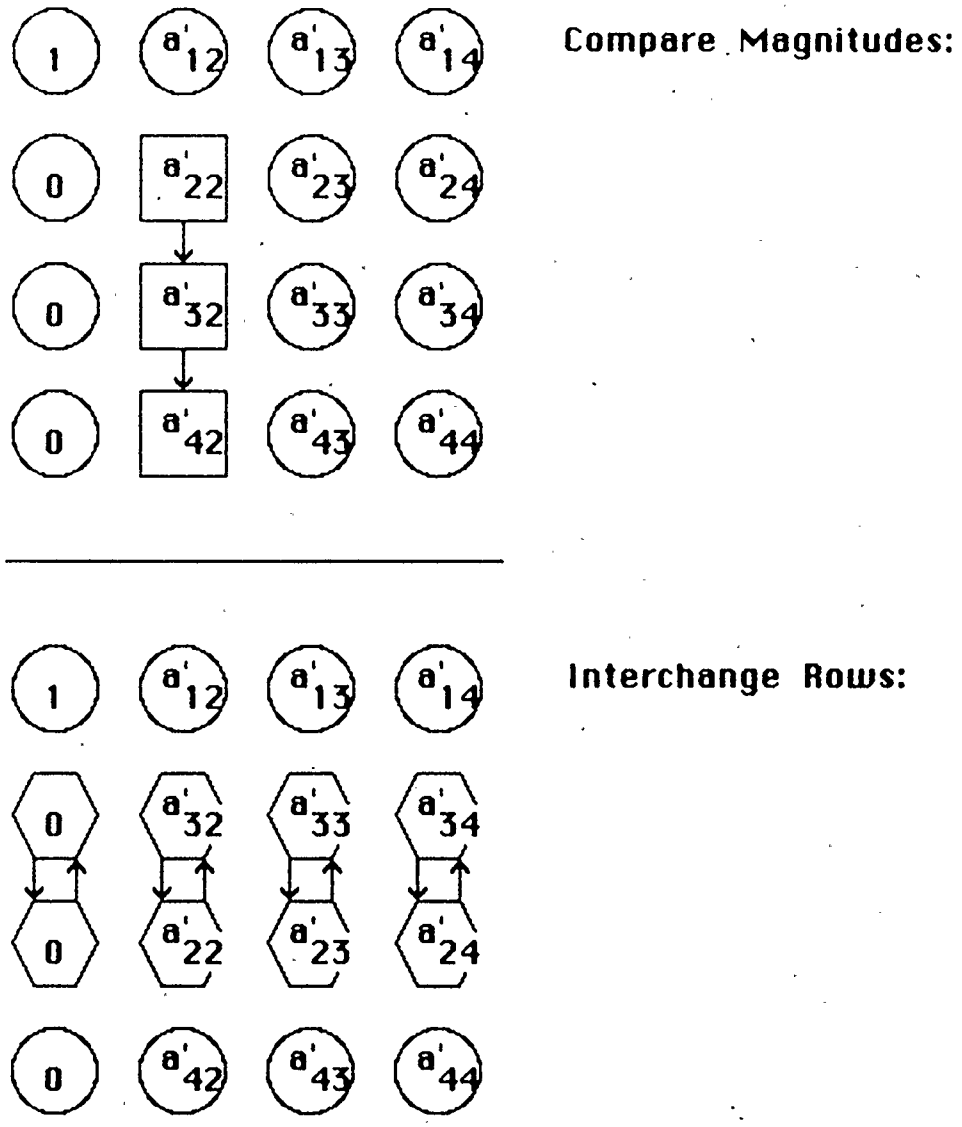
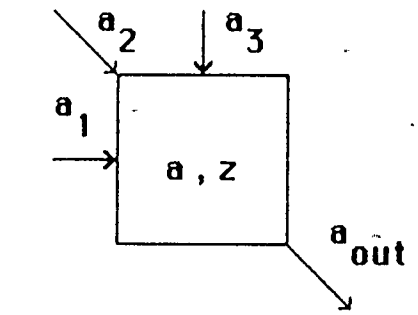


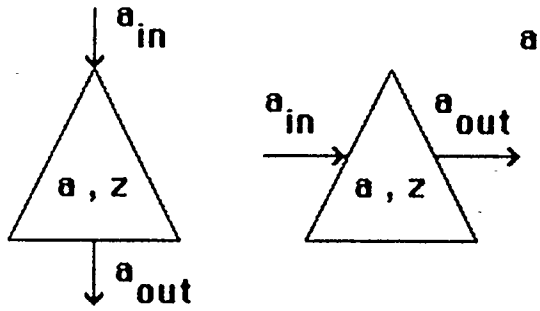
Fig. 3.30: The steps involved in partial pivoting.

TOTAL PIVOTING

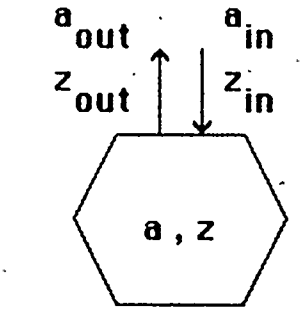


Compare Magnitudes:

$a_{out} = max(|a|,|a_1|,|a_2|,|a_3|)$



$a_{out} = max(|a|,|a_{in}|)$



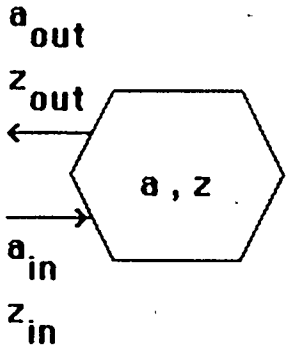
Interchange Rows:

$a_{out} = a$

$z_{out} = z$

$a = a_{in}$

$z = z_{in}$



Interchange Columns:

$a_{out} = a$

$z_{out} = z$

$a = a_{in}$

$z = z_{in}$

Fig. 3.31: Summary of the Total Pivoting algorithm.

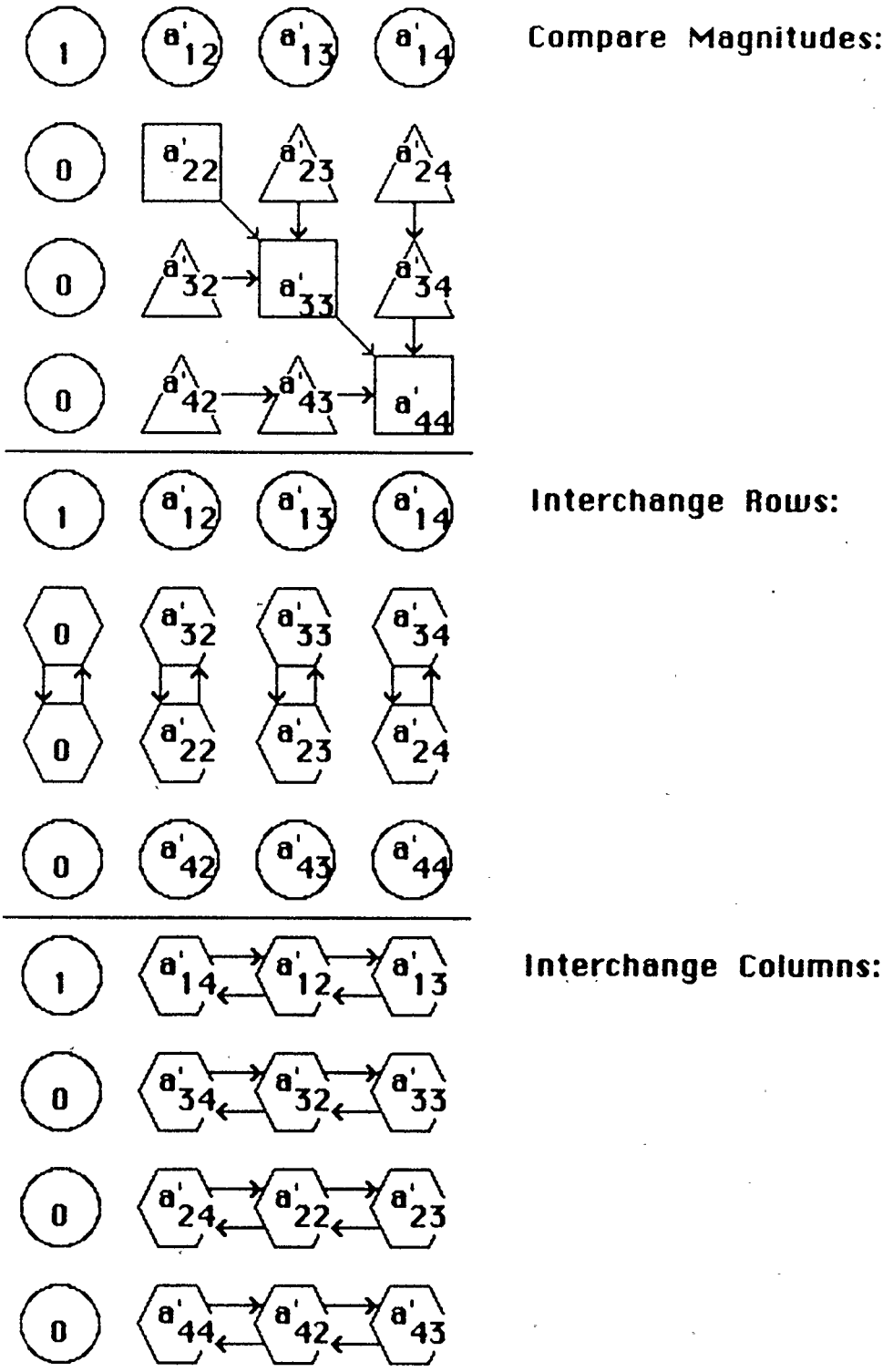


Fig. 3.32: The steps involved in Total Pivoting.

more matrices that are easier to invert. Triangular matrices can be inverted using substitution and thus, a matrix decomposed into a lower and an upper triangular matrix requires only forward substitution and back substitution techniques to obtain the inverse. Calculation of the solution to a system of equations proceeds in the same manner.

The three variations of LU decomposition differ in the way they divide up the original matrix [14]. Crout decomposition yields matrices L and U of the form:

$$[L] = \begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \quad [U] = \begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

Doolittle decomposition gives the following forms:

$$[L] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \quad [U] = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \tag{3.15}$$

Cholesky decomposition yields:

$$[L] = \begin{bmatrix} d_1 & 0 & 0 & 0 \\ l_{21} & d_2 & 0 & 0 \\ l_{31} & l_{32} & d_3 & 0 \\ l_{41} & l_{42} & l_{43} & d_4 \end{bmatrix} \quad [U] = \begin{bmatrix} d_1 & u_{12} & u_{13} & u_{14} \\ 0 & d_2 & u_{23} & u_{24} \\ 0 & 0 & d_3 & u_{34} \\ 0 & 0 & 0 & d_4 \end{bmatrix} \tag{3.16}$$

where the diagonal elements are the square root of each of the pivots. Cholesky decomposition requires the original matrix to be positive definite such that the

square root of each of the pivots exists.

3.5.7 LU Decomposition on a Wavefront Array

It has been shown by Gentleman and H.T. Kung [17] that matrix triangularization can be implemented on a systolic array of the form given in figure 3.33. Rather than using the pivot row to eliminate all the off diagonal elements in a particular column, this algorithm eliminates elements by adding multiples of an adjacent row. The use of "neighbor pivoting" reduces the effects of round-off errors. Rows are interchanged to keep multiples from exceeding unity.

The above technique is intended to be used with back substitution as a method of GE for solving equations. GE requires that equivalent operations be performed on both the original matrix and on all the intended solution vectors at the same time. This requires an array processor of the configuration shown in figure 3.34. Notice that this array is not a convenient shape.

The matrix in its decomposed form is more useful than in its reduced form since any number of solution vectors can be calculated using the L and U matrices. The technique described in the remainder of this section extends the matrix triangularization technique of Gentleman and H.T. Kung to LU decomposition.

The PEs along the main diagonal and the PEs in the upper triangle function exactly as they would during the matrix triangularization algorithm. The

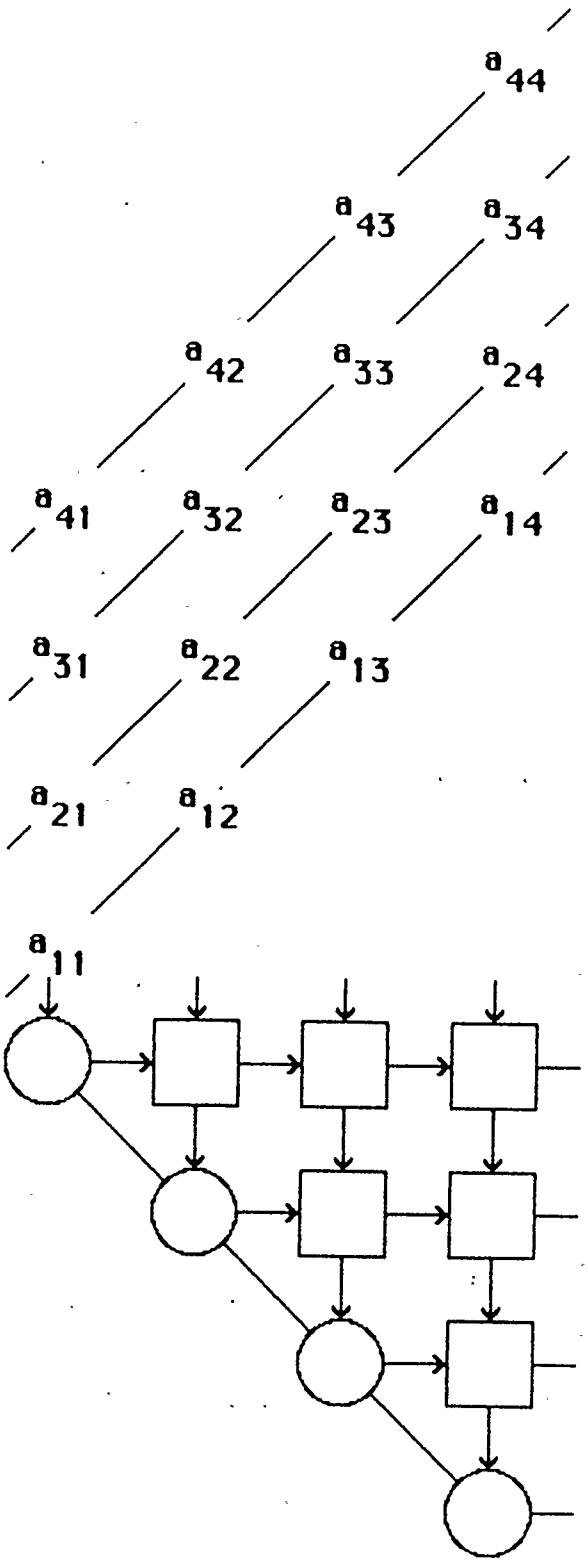


Fig. 3.33: H.T. Kung's systolic array for matrix triangularization.

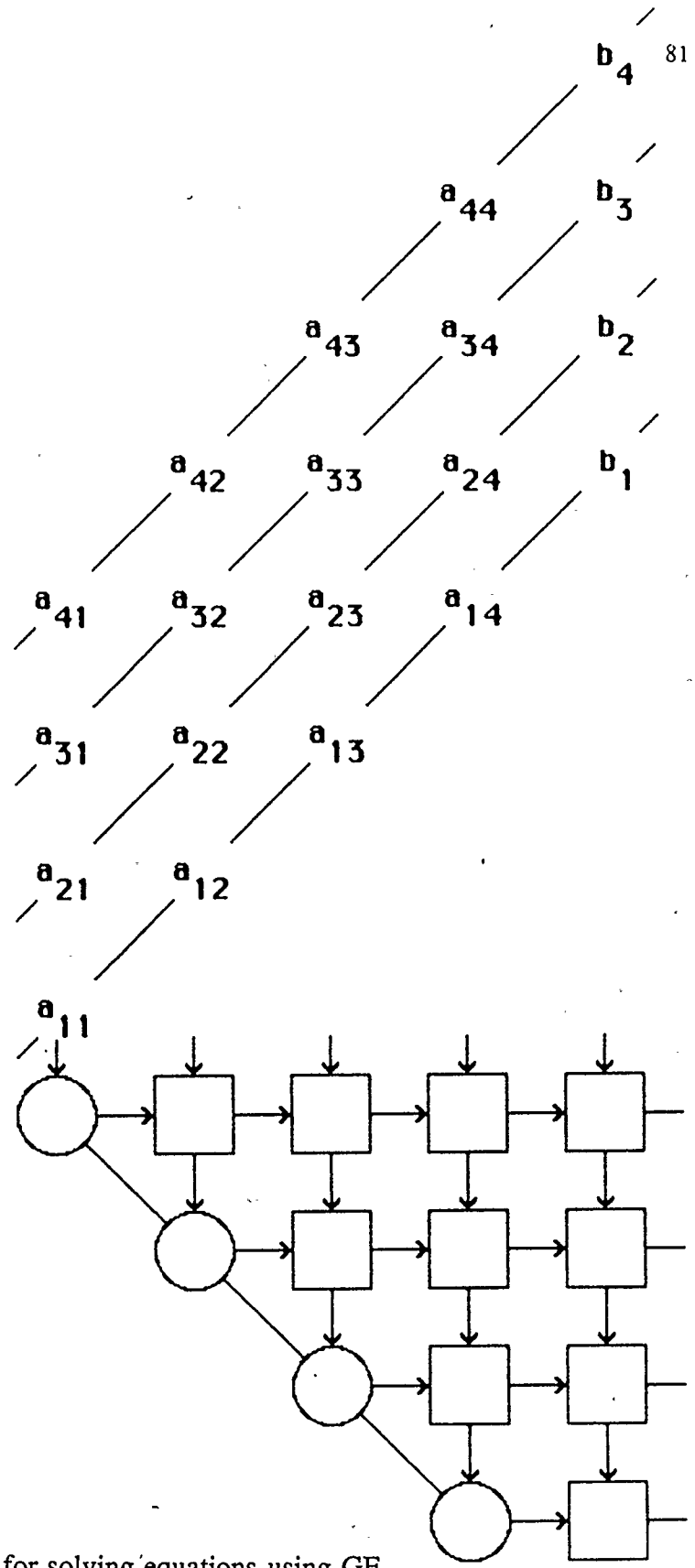


Fig. 3.34: A systolic array for solving equations using GE.

previously unused PEs in the lower triangle of the array are used to store the multiples calculated by the diagonal PEs. This array is shown in figure 3.35 and the function of each type of PE is summarized in figure 3.36. At the completion of the algorithm the elements of matrices L and U reside in the array. Flowing the original matrix into the array from the north produces L and U in the same format as Doolittle decomposition.

Neighbor pivoting can be used with LU decomposition to deal with matrices with a zero along the main diagonal or matrices that are susceptible to round-off errors.

3.5.8 Solving the LU Decomposition

Solving the LU decomposition of a matrix A for a solution vector x such that:

$$[A][x] = [L][U][x] = [b] \tag{3.17}$$

requires solving two simpler equations:

$$[L][y] = [b] \tag{3.18}$$

$$[U][x] = [y] \tag{3.19}$$

where vector y is an intermediate result.

This process is implemented on a wavefront array processor by splitting the array into the portion holding L and the portion holding U. The PEs on the main diagonal hold elements from both matrices so they are involved in the

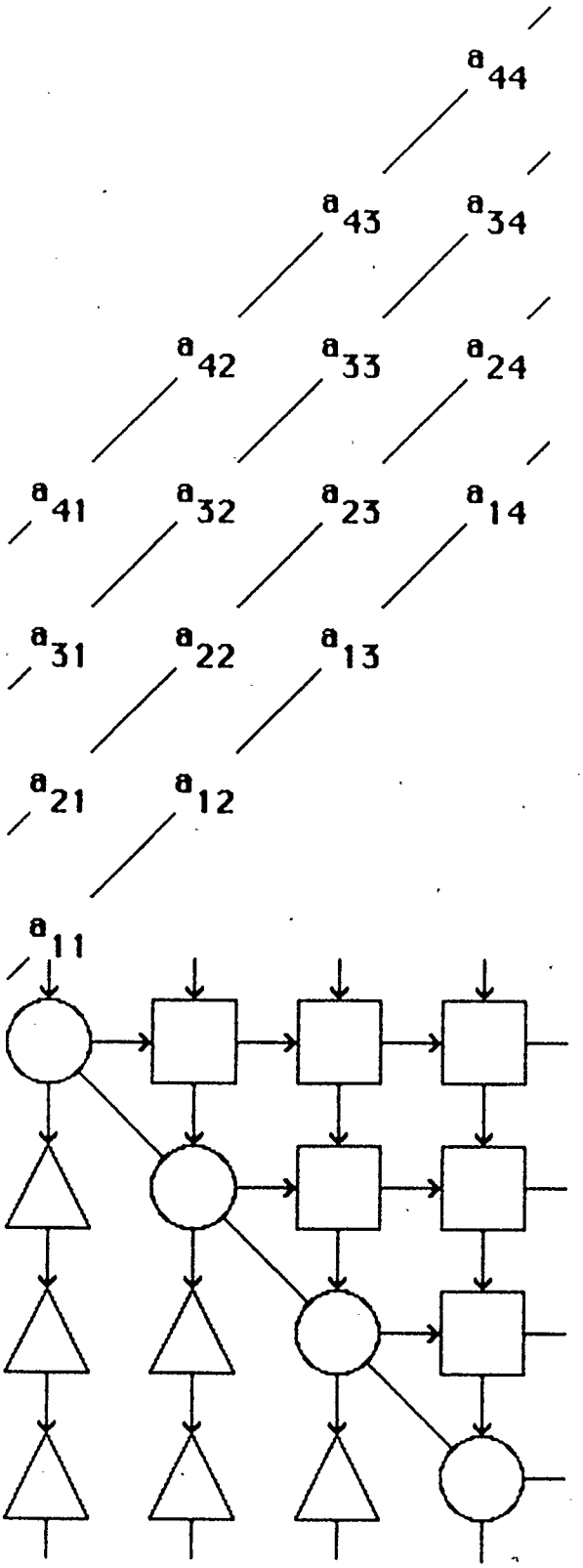
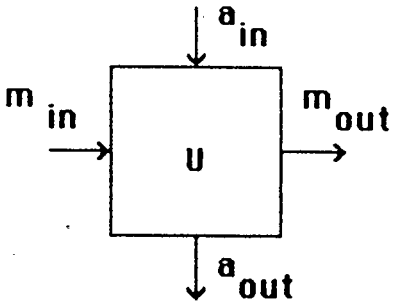


Fig. 3.35: Arrangement of data and PEs for LU Decomposition.

LU DECOMPOSITION

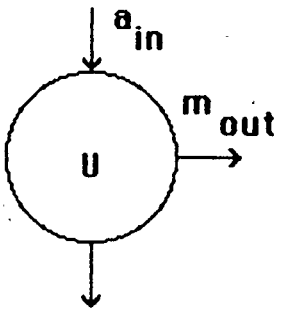


$U = a_{in}$

for all other a_{in} & m_{in}:

$a_{out} = a_{in} - m_{in}$

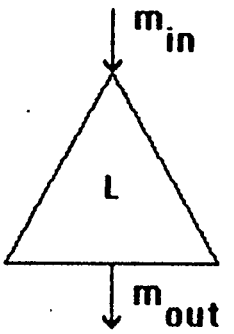
$m_{out} = m_{in}$



$U = a_{in}$

for all other a_{in}:

$m_{out} = a_{in} / U$



$L = \min$

for all other m_{in}:

$m_{out} = m_{in}$

Fig. 3.36: Summary of the LU decomposition algorithm.

solution of both equations. Forward substitution of L using the constant vector b yields a vector y spread along the main diagonal of the array. Back substitution of matrix U using zeros generated by the PEs on the east side of the array results in vector x flowing out the north side of the array.

Figures 3.37 and 3.38 show the arrays for solving for vectors y and x respectively. The algorithms used in solving the equations for vectors x and y are summarized in figures 3.39 and 3.40. Finally, the two algorithms are combined into one as shown by the array in figure 3.41 and the algorithm summarized in figure 3.42.

Any number of vectors can be pipelined into the west side of the array to yield solution vectors provided that the diagonal PEs have the resources to queue up the elements of the intermediate y vectors generated by the lower half of the array until they are used by the top half of the array. Notice that the format of the solution vectors emerging from the north side of the array is not standard. There is no simple way around this inconvenience.

Also note that neighbor pivoting has the undesirable affect of changing the order of the rows in any solution vectors generated using the neighbor pivoted LU decomposition. It is necessary to keep track of the row interchanges made by the neighbor pivoting scheme so that the rows of the solution vectors can be rearranged accordingly. Gentleman and H.T. Kung [17] do not encounter this problem with their triangularization algorithm because GE performs the row interchanges on

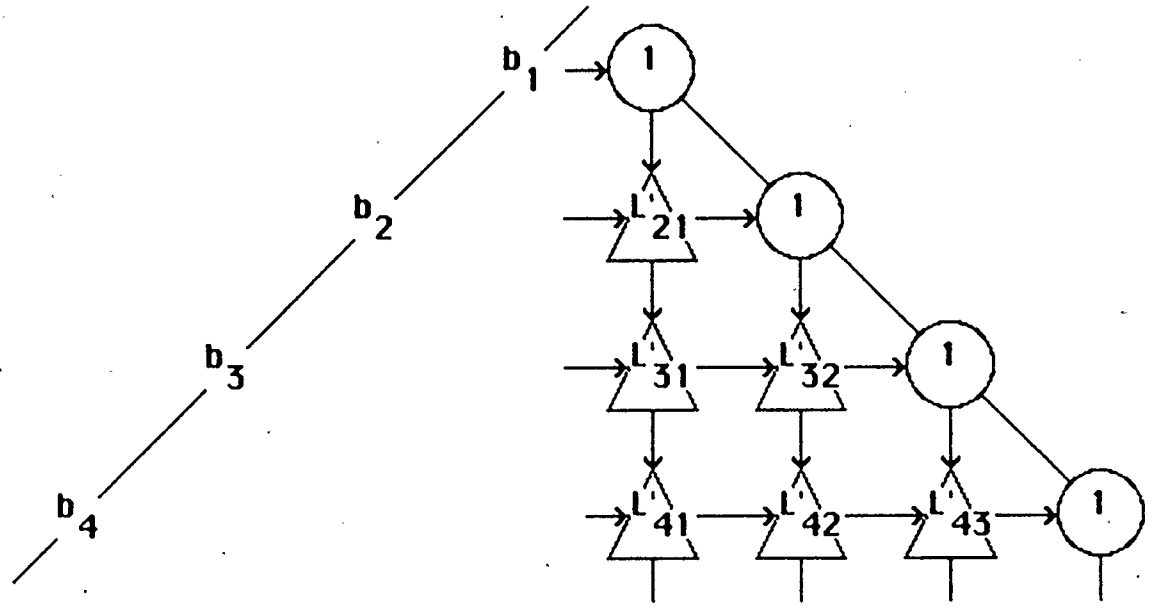


Fig. 3.37: The array for solving $[L][y] = [b]$.

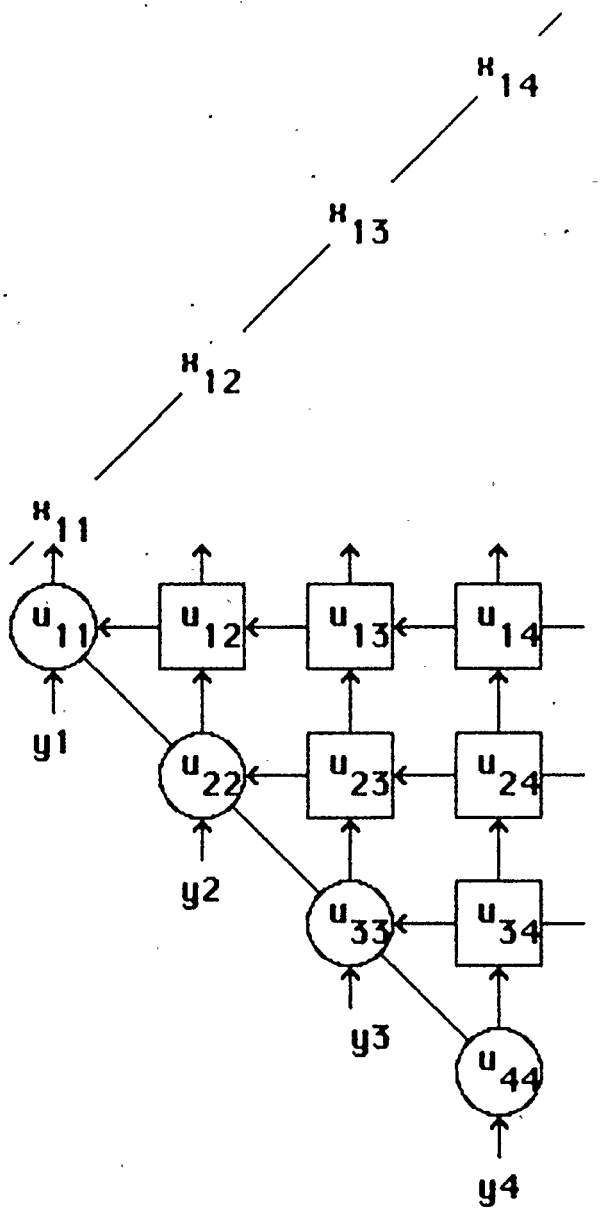


Fig. 3.38: The array for solving $[U][x] = [y]$.

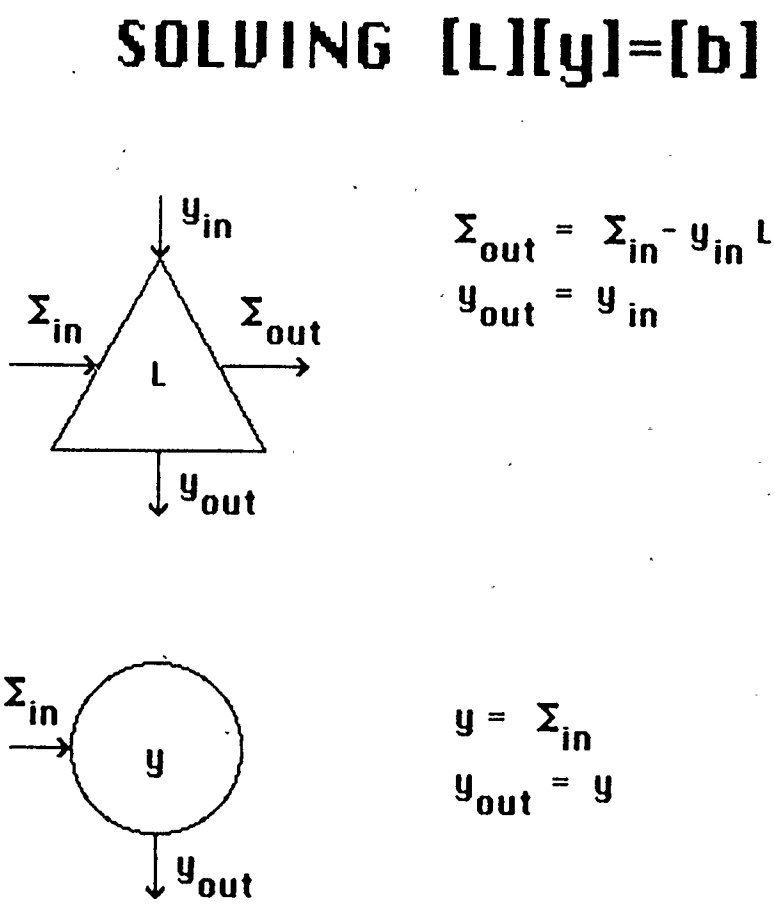


Fig. 3.39: Summary of algorithm for solving [*L*][*y*] = [*b*].

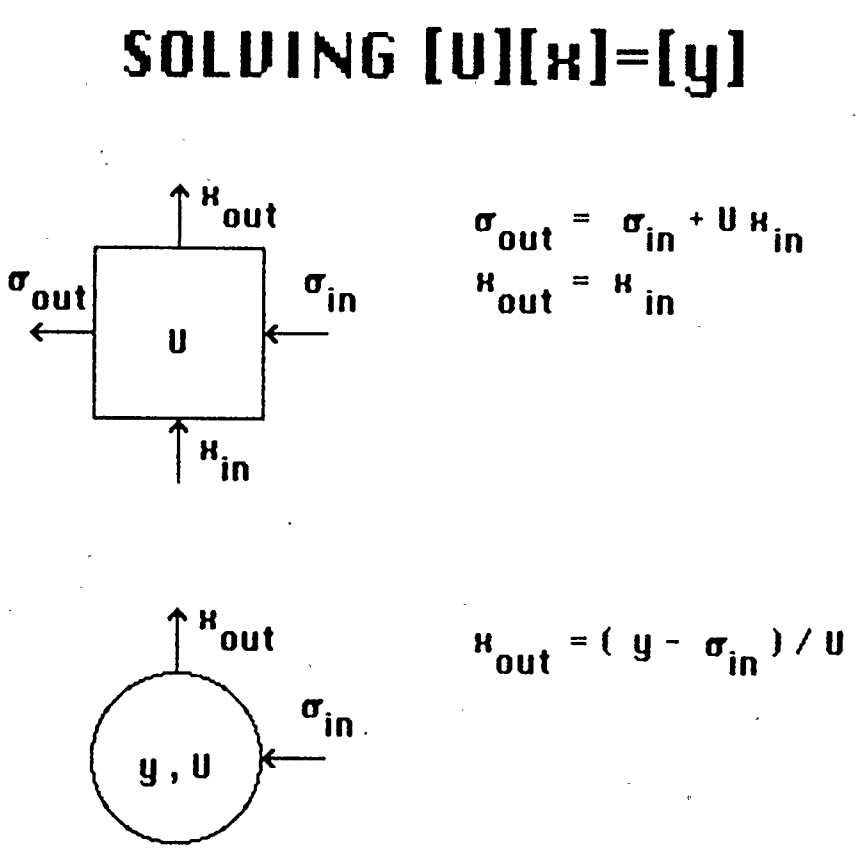


Fig. 3.40: Summary of algorithm for solving $[U][x] = [y]$.

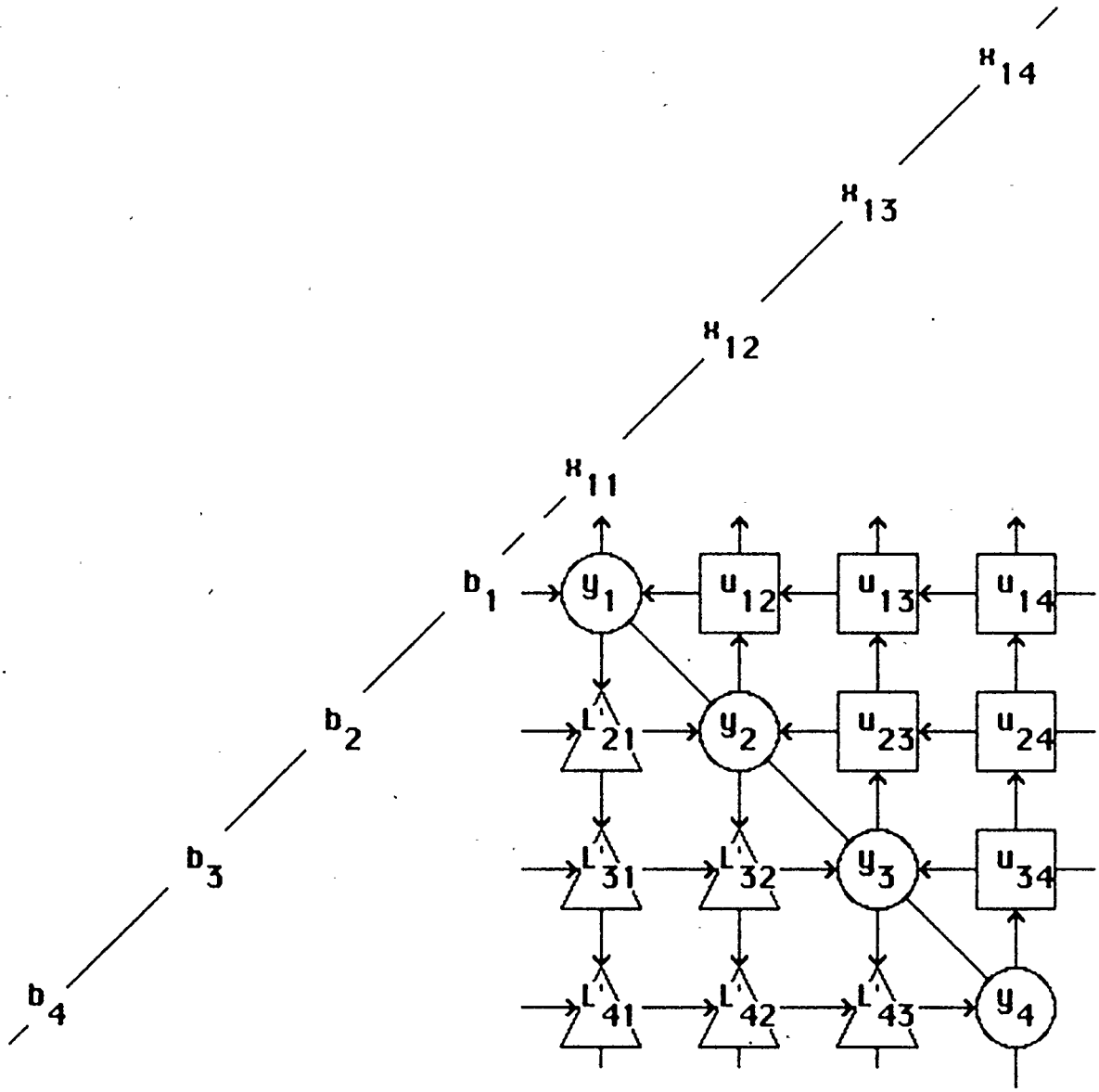


Fig. 3.41: Array for solving $[L][U][x] = [b]$.

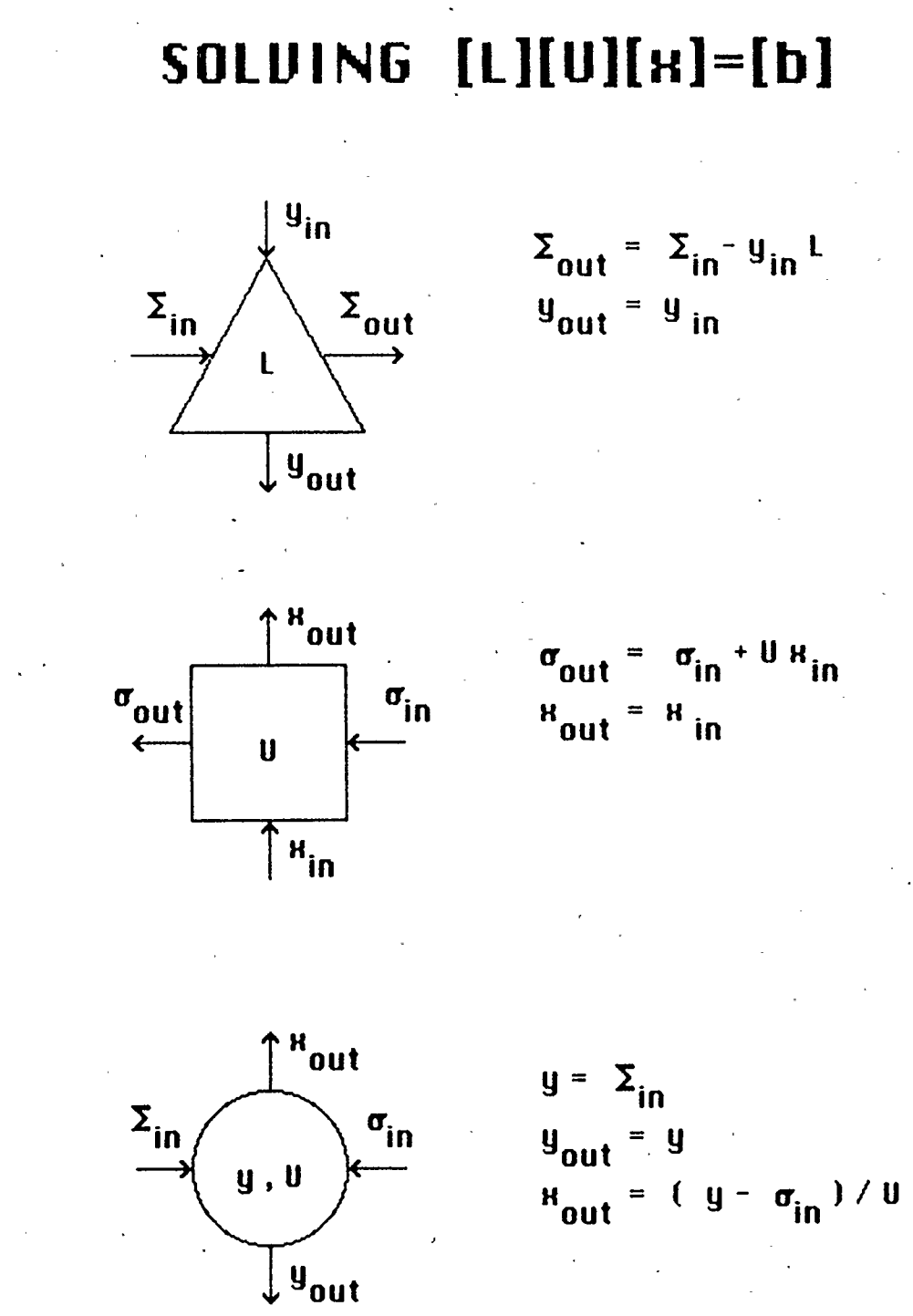


Fig. 3.42: Summary of $[L][U][x] = [b]$ equation solving algorithm.

both the original matrix and on any solution vectors at the same time.

3.5.9 Inverting Large Matrices by Partitioning

Spieser and Whitehouse [18] describe a technique for inverting large positive definite matrices by partitioning them into submatrices. If matrix R is partitioned as shown below;

$$[R] = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{3.20}$$

then the inverse of R is given by;

$$[R] = \begin{bmatrix} [A]^{-1} + [A]^{-1}\,[B][E]^{-1}\,[C][A]^{-1} & -[A]^{-1}\,[B][E]^{-1} \\ -[E]^{-1}[C][A]^{-1} & [E]^{-1} \end{bmatrix} \tag{3.21}$$

where $[E] = [D] - [C][A]^{-1}[B]$.

The cost in computing time due to partitioning is a factor of eight and the total number of operations is: two submatrix inversions and six submatrix multiplications or multiplication-additions compared to one full-sized matrix inversion.

3.6 COMBINATIONS OF ALGORITHMS FOR EVALUATING MATRIX EQUATIONS

There are a number of other matrix equations that can be efficiently evaluated using combinations of the matrix algorithms presented in this chapter. Examples of some matrix equations along with the combinations of algorithms used to accomplish them are given in figure 3.43

$[Z] = [A]([B][C]+[D])$

MULT 2	$([B])([C])$
LOAD	$([D]),([E])$
ADD	$([D])+([E])$
ADD	$([B][C])+([D])$
MULT 1	$([A])(([B][C]+[D]))+[0] \rightarrow [Z]$

$[Z] = [A]+([B][C])^{-1}$

MULT 2	$([B])$, $([C])$
GE	$([B][C])$ with TPIV
LOAD	$([A])$
ADD	$([A])+(([B][C])^{-1})$
UNLOAD	$([A]+(([B][C])^{-1}) \rightarrow [Z]$

$[Z] = [A]([B]+[C]+[D])+[E]$

LOAD	$([B]),([C])$
ADD	$([B])+([C])$
LOAD	$([D])$
ADD	$([B]+[C])+([D])$
MULT 1	$([A])([B]+[C]+[D])+[E] \rightarrow [Z]$

Fig. 3.43: Examples of easily evaluated matrix equations.

Travassos [19] suggests a clever technique for the easy evaluation of the equation:

$$[Z] = [A][B]+[C][D] \tag{3.22}$$

Matrices A and C are combined and so are matrices B and D such that the new equation involves only one matrix multiplication as shown below:

$$[Z] = \begin{bmatrix}[A] & [C]\end{bmatrix} \begin{bmatrix}[B] \\ [D]\end{bmatrix} = [A][B]+[C][D] \tag{3.23}$$

Both of the composite matrices flow through the array so that the size of the array needed to evaluate this equation is not changed.

The matrix equations given in figure 3.44 occasionally occur in applications. These are examples of matrix equations that cannot be evaluated without reloading intermediate results.

3.7 SUMMARY

The algorithms developed for loading, unloading, matrix addition, matrix subtraction, scaling of matrices, matrix multiplication, matrix inversion and solution of systems of equations are consistent with the wavefront concept. These algorithms have the flexibility to deal with operands already residing in the array and operands that must be flowed into the array. Frequently, it is possible to leave intermediate results in the array, eliminating the need to reload them later. This increases the efficiency of the array processor when evaluating matrix equations and eliminates the I/O involved in unloading and reloading. The execution of all

$[Z] = [A][B][C][D]$

MULT 2	$([A])([B])$
MULT 2	$([C])([D])$
UNLOAD	$([C][D])$
MULT 1	$([A][B])([C][D])+[0] \rightarrow [Z]$

$[Z] = [A][B]^{-1}[C]$

LOAD	$([B])$
GE	$([B])$ with TPIV
MULT 1	$([A])([B]^{-1})+[0]$
UNLOAD	$([A][B]^{-1})$
MULT 2	$([A][B]^{-1})([C])$
UNLOAD	$([A][B]^{-1}[C]) \rightarrow [Z]$

$[Z] = ([A][B][C])^{-1}$

MULT 2	$([A])([B])$
MULT 1	$([A][B])([C])+[0]$
LOAD	$([A][B][C])$
GE	$([A][B][C])$ with TPIV
UNLOAD	$(([A][B][C])^{-1}) \rightarrow [Z]$

Fig. 3.44: Matrix equations that require reloading intermediate results.

wavefront algorithms can be overlapped to increase the rate that matrix equations are evaluated by keeping all the PEs busy as much of the time as possible.

GE and LU decomposition are relatively fast and efficient operations. However, partial and total pivoting add many extra steps to each step of GE and are difficult to implement. Neighbor pivoting, used in the LU decomposition algorithm, results in the need to keep tack of row interchanges since these affect the order of the rows in the solution vectors.

Combinations of these wavefront algorithms can be combined and overlapped to allow efficient evaluation of a number of more complicated matrix equations. All matrix equations can be evaluated using the set of algorithms described in this chapter.

Chapter 4

IMPLEMENTATION OF THE ALGORITHMS

4.1 IMPLEMENTATION: HARDWARE VERSUS SOFTWARE

The ultimate implementation of the array processor is the fabrication of a set of chips using VLSI techniques. However, VLSI implementation is a time consuming and error prone task that cannot be justified until confidence in the design has been established. Verification of the algorithms presented in the Chapter Three, and the estimation of their performance, requires some other form of implementation.

The design constraints and concepts for systolic and wavefront array processors were formulated with VLSI implementation in mind. A VLSI array processor would give maximum performance and only when such an array is made can total success be declared. However, the anticipation of shortcomings in algorithm and array designs suggests that an inflexible implementation is unwise. Fischer et al [20] discuss their experiences during a four man-year VLSI implementation of a programmable PE involving the complexity equivalent to a few hundred LSI chips. The programmability of their design gave the PE the flexibility to be used in a number of different applications. However, the penalty of this versatility was complexity. Their success was limited by the lack of sophisticated

design tools and experience required for projects of this complexity.

Hardware and software implementations differ in: the amount of work needed to achieve the implementation, the ease of making changes, and how closely the operation of the algorithms and the array are simulated. Some of the alternative forms of implementation are considered below.

A hardware implementation using programmable microcomputer chips facilitates greater ease in making corrections and changes to the algorithms and the array operation. The hardware approach has been used by Symanski [21] who has built an eight by eight array with reconfigurable interconnections between PEs and developed software to allow programming of the PEs using a high level language. Each PE is comprised of a single chip microcomputer (Intel 8031) with an arithmetic processor (Intel 8231) and a collection of other standard TTL ICs. Unfortunately, this approach is expensive. Symanski puts the cost at about two hundred and fifty dollars per PE. Progress reports [22, 23] indicate that implementations of matrix multiplication and singular value decomposition algorithms have been achieved.

A software implementation uses a high level programming language to model the array processor and to simulate the algorithms. This allows more flexibility to make design changes than hardware approaches and does not incur the long development time or the heavy price tag. Flanagan and Woo [24] recommend the use of software simulation in verifying parallel processing systems. The use of

a hierarchical structure in the design of the simulator allows the simulations to follow along with the top-down style commonly employed in system design.

The software simulation approach was used to verify the operation of the array processor and the algorithms presented in Chapter Three. The PEs were modeled and connected together into an array. The PEs were programmed to perform each of the different algorithms. A step by step display of the array of PEs carrying out the algorithms permitted verification that the array processor and the algorithms worked together to give the expected results. The programming language used to write the simulator is introduced in the next section, followed by explanations of the major problems encountered in simulating the array processor. Finally, the results of the simulations with respect to algorithm performance are reported.

4.1.1 Discrete Event Modeling On Simula: DEMOS

DEMOS [25] is an extension to the general purpose programming language Simula. DEMOS provides a context for simulation work with facilities for defining entities and resources that model real-life processes and objects, as well as providing automatic scheduling of processes, collection of data, tracing of events, and generation of reports.

The use of DEMOS requires knowledge of only a subset of Simula and substantially reduces the amount of simulation code to be written.

DEMOS takes the process approach to simulation [26]. This allows the code which describes how an entity performs an operation such as how a PE performs matrix addition, to be written as a subroutine. Similar subroutines are used to describe how a PE performs matrix multiplication, inversion etc. The code closely follows the algorithm so that development and debugging of the code takes as little time as possible. This close similarity is important since the motivation for doing the simulations is to check for mistakes in the design of the algorithms. It is essential to have confidence in the representation of the array and the algorithms in order that negative simulation results can be traced to deficiencies in the algorithms rather than to mistakes in the simulation code.

4.1.2 Modeling of the Processing Elements

Both diagonal and off-diagonal PEs are defined in the simulation code to be sub-classes of a general processor definition shown diagrammatically in figure 4.1. Each processor has:

-a random access memory (RAM) for storing elements of different matrices,

-a stack and queue for the temporary storage of data,

-an instruction register to indicate which matrix operation the PE is performing,

-a parameter area for storing the parameters associated with each matrix operation,

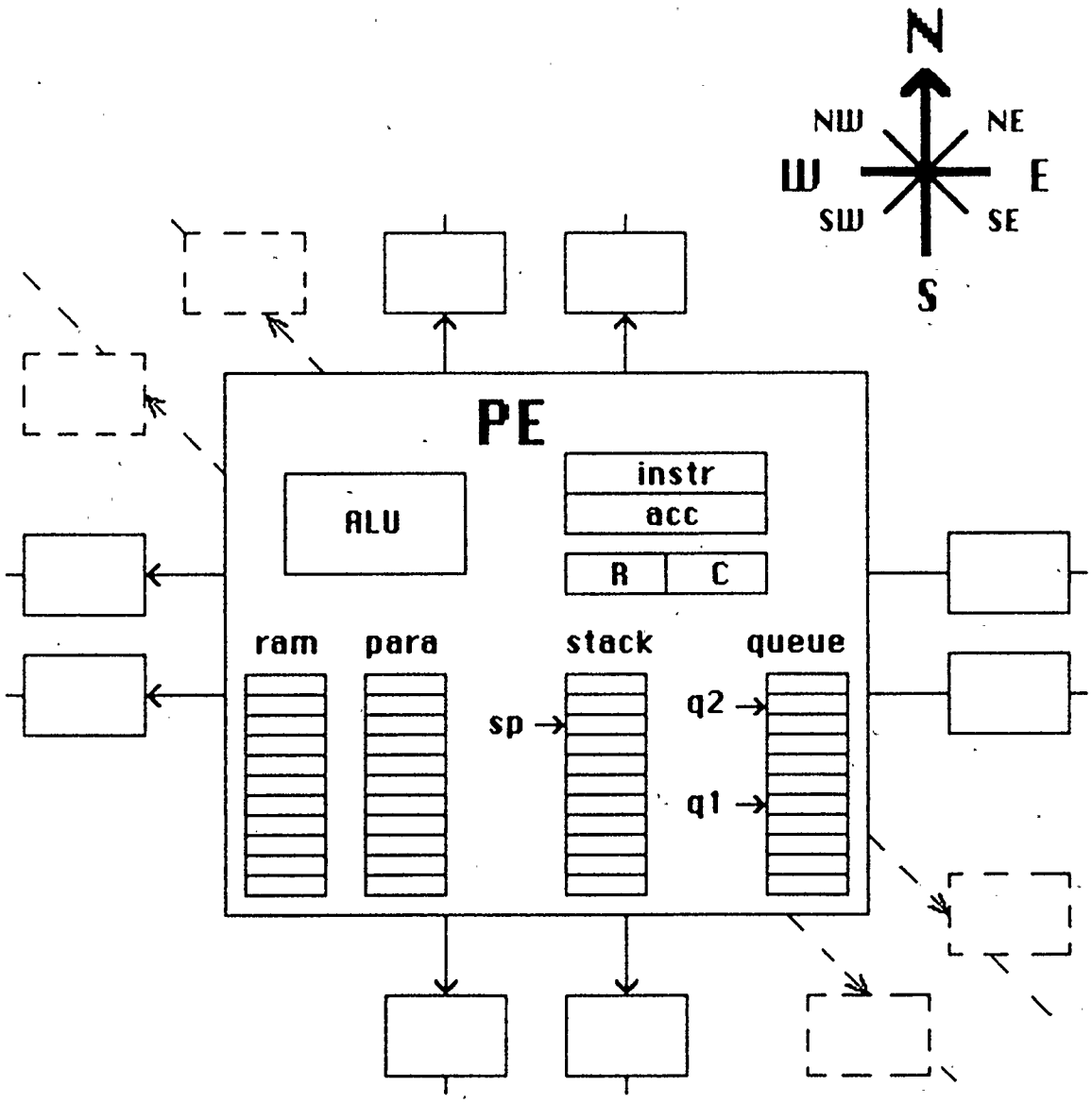


Fig. 4.1: A generalized model for a PE.

-an accumulator to display the result of each step,

-an arithmetic logic unit (ALU) to perform arithmetic operations.

Buffers hold the information sent by a PE until another PE is ready to receive the information. One set of the buffers holds the information sent by a particular PE and another set holds the information that the PE is to receive. These buffers are accessed by means of pointers which are named according to the direction in which the particular buffer is located relative to the PE.

A number of primitive functions which each processor can perform is included in the PE definition. This set of primitives includes:

LookINSTR - returns boolean true if the specified buffer contains an instruction.

LookDATA - returns boolean true if the specified buffer contains data.

WaitINSTR - waits as many steps as needed until the specified buffer receives an instruction.

WaitDATA - waits until the specified buffer receives data.

GetINSTR - returns a copy of the instruction residing in the specified buffer. Resets flags for that buffer to indicate that the information has been read.

GetDATA - returns a copy of the data residing in the specified buffer. Resets flags for that buffer to indicate that the information has been read.

FindINSTR - returns a pointer to the first buffer that contains an instruction.

FindDATA - returns a pointer to the first buffer that contains data.

SendINSTR - puts the instruction in the specified buffer as soon as the buffer is empty. Sets flags to indicate buffer is full and contains an instruction.

SendDATA - puts the data into the specified buffer as soon as the buffer is empty. Sets flag to indicate the buffer is full.

PARAMETERS - copies the encoded parameter from a predefined buffer and decodes it into the specified number of parameters. The parameters are put into the parameter memory. Passes the encoded parameter along to other PEs.

INSTRout - sets flags to indicate that the specified instruction is to be passed on to other PEs.

INSTRhandler- sends any instructions which are to be passed during a particular step.

The simulation code distinguishes between diagonal and off-diagonal PEs. The definition for each type of PE includes the code for each of the matrix operation algorithms and a main program which receives new instructions and executes the routine that implements the specified matrix operation. The definitions for the two types of PEs differ in the way that certain matrix operations are performed and by the inclusion of extra buffer pointers to model the interconnections between diagonal PEs. These diagonal interconnections are shown in figure 4.1 by dotted lines.

The PE model described here is modest by present microcomputer

standards. But, at the same time, the model contains more resources than are usually found in the systolic arrays described in recent literature. These extra resources, namely the memory capable of holding elements from several different matrices, the stack and the queue, are justified by the low expense that they incur in the design of the PE. The extra memory reflects the desire to handle intermediate results efficiently. Often this means keeping matrices in the array for later use to avoid the need to reload them. The concept of this "vertical memory", (that is, memory in the third dimension above the two dimensions occupied by the array) was suggested by Speiser and Whitehouse [18] to simplify operations that involve partitioned matrices.

4.1.3 Connecting Processing Elements to Form an Array

An array of PEs is created when instances of the two types of PEs are joined together. The simulation code sets the buffer pointers of the PEs to the proper buffers in order to simulate the interconnections between PEs.

Ports around the outside of the array pass data and instructions in and out of the array. Surrounding the array with intelligent ports allows all the PEs to be treated the same, even the PEs located on the boundaries of the array.

4.1.4 Initializing the Array

It is particularly useful to have each PE know its coordinates within the array. This knowledge allows a PE to determine if it will be active during any

matrix operations which involve matrices smaller than the size of the array. The coordinates, together with the parameters associated with an instruction, allow a PE to calculate the number of times a sequence of operations needs to be performed. Frequently, the number of repetitions of a recursive algorithm depends on the position of the PE.

Implementation of the LOAD algorithm is a good example of why PEs should know their position in the array. In the LOAD algorithm, each PE takes the first matrix element it receives and passes on all the others. This rule is easily stated but not as easily programmed, because some mechanism is required to indicate that all of the matrix elements have been received and that the LOAD operation is complete. There are undoubtedly many ways to provide this mechanism, several of which are discussed below.

The number of matrix elements to be received by each PE could be passed as a parameter, immediately following the LOAD instruction. However, since the number of elements decreases by one for each column in the array, this parameter must sometimes be altered as it is passed through the array. The decision to decrement the parameter depends upon the direction the parameter is to be passed. This operation is therefore not straight forward and the logic needed to make this decision is time consuming and code intensive.

Another possible solution is to precede each input data stream with a number indicating the number of matrix elements that follow. Since data is only

passed in one direction, unlike parameters that are passed three directions, the number must be decremented by every PE before it is passed on. Alternatively, an "end of data" token could follow the matrix elements into the array. The disadvantages of these alternatives are:

-they waste a data wavefront for the number or for the token,

-some way of inserting the numbers or tokens in with the matrix elements is required.

A third possible solution is to have each PE check for a new instruction after each matrix element is passed on. A new instruction will not be received by a PE until the current instruction is complete. Unfortunately, the next instruction is always waiting for PE(1,1). PE(1,1) would have to become a special case or else the instruction port must hide instructions from PE(1,1) until the proper time. The instruction port has no way of knowing when the current instruction is done and so the original problem reappears.

Occasionally nasty problems have relatively nice solutions. The answer to the above problem is to pass the size of the matrix to be loaded into the array as a parameter but not to alter the parameter as it is passed between PEs. If each PE knows its position in the array then the number of matrix elements that it must pass on after keeping the first element is equal to the size of the matrix minus the column number of the PE.

The solution of this awkward implementation problem becomes almost trivial when the PEs are informed of their positions. This same technique applies to the implementation of UNLOAD as well. All of the implementations of the matrix algorithms are simplified by possession of this knowledge.

The purpose of the initialization routine is to inform each PE of its position within the array. Coordinates initially set to zero are passed through the array as parameters. As the coordinates are passed between PEs, they are updated to represent the position of each PE. The PEs record their coordinates for future reference.

4.1.5 The Array Processor in Action

The contents of the array processor are displayed after each step of an algorithm is executed. The information provided by the display routine is usually sufficient to verify the correct operation of the algorithm and the array. The format of the display is shown in figure 4.2. This format is similar to the style of the diagrams presented in the previous chapter which also give a step by step illustration of the array processor in action. The display shows the positional relationships of PEs and ports, showing the current contents of each and the instruction being executed by each PE.

A detailed display of the contents of any number of specific PEs can be obtained after any step. This display shows the contents of all the registers and

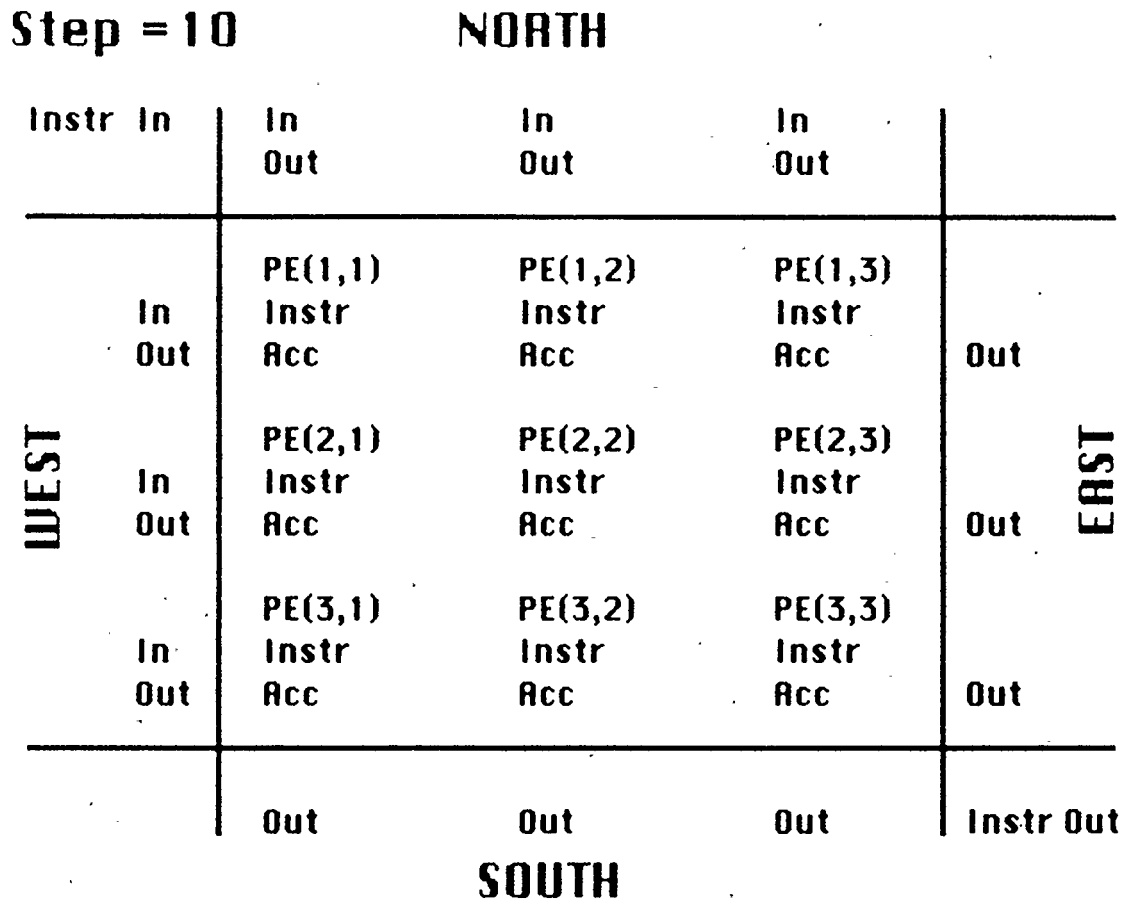


Fig. 4.2: The display of the array processor provided by the simulator.

memory in a particular PE. Figure 4.3 shows the form of the detailed display.

4.2 PROBLEMS ENCOUNTERED IN THE SIMULATIONS

4.2.1 Simulation of Concurrent Processes

The simulation of a parallel processing system on a computer utilizing only a single processor requires a mechanism to fake concurrency. This section describes how DEMOS handles the simulation of concurrency.

DEMOS uses an event list to keep track of when the actions of entities are scheduled to occur. When an entity is scheduled, it is merged into the event list in a position after the entities whose actions are scheduled to occur during the same step or sooner. This rule maintains the ordering of the event list in accordance with the time of the next scheduled action.

The actions of the entity at the head of the event list are executed. Once its actions are completed, the entity can be rescheduled or terminated. The actions of the next entity in the event list are then executed. The rescheduling of actions can be used to represent a delay, such as the amount of time needed to perform some action, or it can represent the stepwise nature of a process. The "hold" mechanism in DEMOS accomplishes a delay of known length by reinserting the entity in the event list in the appropriate position. Thus, an entity can perform some action, wait a fixed number of steps or length of time, and then resume its

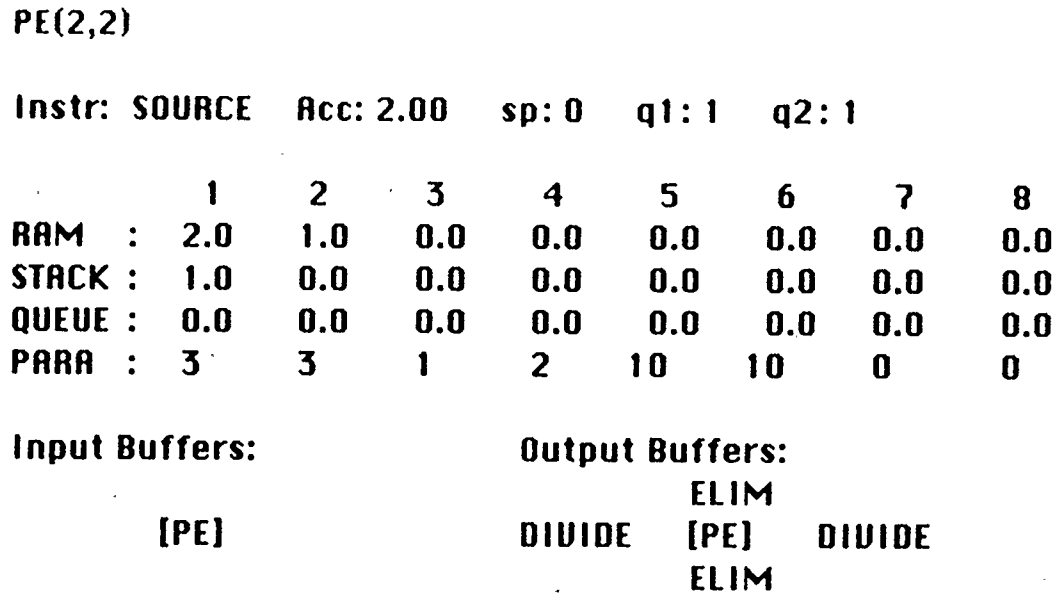


Fig. 4.3: Detailed display of a PE.

actions.

All the entities scheduled to perform actions simultaneously appear in the event list together. The actual order of this part of the event list should not have any influence on the outcome of the simulation if concurrency has been simulated properly. There are situations where unexpected problems do arise however. Consider the case where an entity A is to send entities B and C a message. Suppose these three entities appear in the event list in the order A, B, C. Then A will send the message, B will receive the message and C will receive the message all during the same step in the simulation. Now consider a different ordering, for example, B, A, C. Entity B will not receive the message from A but entity C will. This situation illustrates the need for a further mechanism to simulate this type of process interaction properly.

In a self-timed system such as purposed by the concept of a wavefront array processor, it is the passing of information that synchronizes the operation of the system. The situation described above clearly does not give results consistent with the results expected for such a system.

A simple solution to this difficulty is to write the code for the communication of information such that all of the entities have the opportunity to receive messages before any of the entities is allowed to send information. Alternatively, the reception of messages could be delayed until after all entities have had a chance to send.

This affects the simulation code describing the actions of the PEs during each step of the different algorithms. In the context of the movement of wavefronts, the term "step" refers to the movement of a wavefront between adjacent PEs, so that a "step" in an algorithm can be defined as all the actions that a PE performs from the time a wavefront is received until it is sent. Thus, to properly simulate an algorithm, each step of the algorithm must take the form shown below:

1) Get information,

2) hold (until all other PEs have performed step 1),

3) Pass along information contained in wavefront,

4) Use information to calculate results,

5) Send results,

6) hold (until other PEs have completed steps 3-5).

Not all the above parts are compulsory; there may be no calculation associated with some step of an algorithm, or perhaps the PE requires no information in order to carry out a step.

The use of the "hold" mechanism provided by DEMOS helps to solve the difficulty of simulating concurrency, and it has the additional benefit of imposing structure and consistency to the code describing the steps of each algorithm.

The format of step described above was adopted to solve a simulation problem, but it is not necessarily the most efficient format of step for a real-life

implementation of the array processor. While it is advantageous to maintain some format, for consistency and structure, the order of operations is different for a real system. In such a system, it is best to send information as soon as possible such that the delay of information in passing through PEs is minimized and the speed at which wavefronts pass through the array is maximized. In real-life there is no possibility of data arriving before it is sent, so that there is no reason for holding the actions of a PE while other PEs reach a certain point, this synchronization is inherent in the operation of the system. Thus, the "real-life step" would be of the format shown below:

1) Get Information,

2) Send Information,

3) Do calculations,

4) Send results of calculations,

5) Begin next step when next wavefront arrives.

It is worthwhile to design algorithms so that each step in an algorithm involves approximately the the same amount of work, particularly with respect to any calculations that need to be performed. Consistency in the workload of each wavefront maximizes the array efficiency by minimizing the amount of time that one wavefront delays the propagation of another wavefront. The speed at which wavefronts propagate through the array will tend to the speed of the slowest wave-

front.

4.2.2 Parameter Passing

The variety of parameters associated with the different matrix operation algorithms extends from integers representing the dimensions of matrices, to pointers to ram locations where matrices are stored, to boolean values representing whether options are to be invoked. The number of parameters varies from one to six depending on the particular matrix operation. It would be wasteful to pass each parameter in a wavefront all by itself since more time would be spent passing parameters than performing algorithms. Instead, all of the parameters are encoded into a single wavefront.

The format of the parameter wavefront is similar to the format of a microword used in microprogramming. Each parameter is assigned to a particular field with several parameter fields being accommodated in one wavefront. This packing of a wavefront is possible because parameters are either booleans or small integers. Use of fields that are all the same size simplifies the extraction of parameters from the wavefront. Each algorithm includes a call to a parameter handling routine which extracts the parameters and places them in the parameter memory locations.

4.2.3 The Typical Format of Code for Algorithms

Experience in writing the simulation code for the different algorithms

has shown that the code usually conforms to the format shown in figure 4.4.

The matrix inversion algorithm differs slightly from this form because it makes use of several sub-instructions. The inversion routine follows along the lines of the processor main program which simply finds and executes sub-instructions. The sub-instructions take the form shown in figure 4.5 where each step has the usual format of receive, hold, pass, calculate, and send.

4.2.4 A Provision for Short Instructions

A special provision is needed to cope with the possibility that some instructions are only one step in length. One step instructions are a problem because *passing instructions along the main diagonal of the array is a two step process*. Recall that diagonal processors pass along the newly received instruction east and south during their first step and then southeast on their second step. If all instructions were required to be at least two steps in length then passing instructions could be coded for each algorithm as shown in figure 4.6.

Anticipating the possibility of one step instructions greatly complicates the process of passing instructions because of the situations described in figure 4.7.

It is undesirable to complicate every algorithm by the logic needed to handle instructions in these different situations. A routine called "INSTRhandler" takes on this task and includes all the logic to decide which instructions need to be passed and what directions they need to go. Proper handling of instructions is

```
procedure TYPICAL_MATRIX_OPERATION
begin

procedure DATA_in
begin
  Routine for obtaining the data needed to carry
  out the calculations for each step
end

procedure DATA_out
begin
  Routine for sending the results of the calculations
  for each step of the algorithm
end

Initialization
End_of_step
PARAMETERS(# of parameters);
End_of_step
Check if this PE is within the active part of the array
while #repetitions>0 do
 begin
   DATA_in
   Wait for other PEs to get their data
   Call the Instruction handling routine
   Perform_calculations
   DATA_out
   End_of_step
 end
end
```

Fig. 4.4: The typical format for the code implementing wavefront algorithms.

```
procedure INVERSE_sub-instruction
begin
STEP one
 . . .
STEP two
 . . .
STEP three
 . . .
etc
end
```

Fig. 4.5: The format of sub-instructions.

```
STEP one:
  Get data
  Hold
  SendINSTR(East)
  SendINSTR(South)
  Do calculations
  Send results
End_of_step
STEP two:
  Get data
  Hold
  SendINSTR(Southeast)
  Do calculations
  Send results
  End_of_step
STEP three;
  Get data
  Hold
  Do calculations
  Send results
End_of_step
etc
```

Fig. 4.6: Instruction handling code included in a multi step instruction.

Case 1: One Step Instruction (OSI) followed by a Two Step Instruction (TSI);

Step one
 Send OSI east and south
Step two
 Send OSI southeast
 Send TSI east and south
Step three
 Send TSI southeast

Case 2: One Step Instruction followed by several other One Step Instructions;

Step one
 Send OSI 1 east and south
Step two
 Send OSI 1 southeast
 Send OSI 2 east and south
Step three
 Send OSI 2 southeast
 Send OSI 3 east and south
STEP four
 Send OSI 3 southeast
 Send OSI 4 east and south
etc

Fig. 4.7: Cases that complicate passing instructions between PEs.

assured provided that the INSTRhandler is called sufficiently often so that it can send out whatever instructions need to be passed on. To be safe, it should be called during every step. The routine quickly decides if no instructions need to be sent so that little time is wasted.

For consistency, an INSTRhandler routine was also written for the off-diagonal processors. This routine is quite straightforward since this is a one step process but its existence results in the matrix operation subroutines for both types of PEs being identical for nearly all of the algorithms. Thus, the code for the implementation of these matrix operations only needs to be written, tested and debugged once. This is a very enviable situation since most of the differences between the two types of PEs are isolated to the INSTRhandler routines.

4.3 SIMULATION RESULTS

This section summarizes the results of the simulations and discusses their implications on the array processor performance. The simulation results are compiled in table 4.1.

The first column of table 4.1 identifies the matrix operation and the sizes of the matrices or vectors used as operands. The rest of the results are based on matrices and vectors of these dimensions.

The second column gives the number of parameters associated with

OPERATION (dimensions)	# of PARA	# of DATA	# of steps total	# of steps overlapped	# of calc. steps	Hardest step
LOAD (mxn)	6	n	2n+m+1	2n+1	0	
UNLOAD (mxn)	5	n	2n+m+1	2n+1	0	
ADD SUB (mxn)+(mxn)	5	0	n+m+1	2	1	x=a+b
SCALE (mxn)	3	1	n+m+2	3	1	x=sa
MULT 1 (mxn)(nxp)	4	n	m+n+p+1	n+2	n	x=x+ab
MULT 2 (mxn)(nxp)	4	n	m+n+p+1	n+2	n	x=x+ab
LU (nxn)	3	n	3n+1	n+2	(1) n-1	x=x-ma,m=a/b
SOLVE (nxn)(nxm)	4	m	4n+m+1	m+2	m	S=S-Ly
GE (nxn)	5	0	6n+1	5n+1	n	a=a-mx,m=a/b z=z-mx

(1) Number of calculations depends on position of PE.

Table 4.1: Summary of the simulation results.

each of the algorithms. These numbers emphasize the need to encode the parameters into a single wavefront. Otherwise, the number of parameter wavefronts would be comparable to the number of data wavefronts (given in column three) and would double the number of steps needed to perform the matrix operation. The parameter and instruction wavefronts represent the overhead required to give the array processor instruction set flexibility to handle different matrix operations and different sizes of matrices.

The number of steps taken to perform the various algorithms are shown in the fourth column. These values give an indication of the response time of the array when performing each of the algorithms. These results are based on the assumption that the array dimensions are n by n. The response time is increased as the size of the array is increased. Therefore, the size of the array should not be larger than the maximum size of the matrices that are used in the calculations.

More than one matrix operation is usually required to evaluate a matrix equation. Notice that the total time taken to evaluate the equation is not the sum of the number of steps for the individual matrix operations because the execution of wavefront algorithms can be overlapped. This substantially reduces the total number of steps. Column six gives the number of steps required to perform the algorithm when its execution is overlapped with the execution of the instructions before and after it.

The two remaining columns in table 4.1 give the number of calculation steps in the different algorithms and give an example of the calculations required during the most computationally intense step.

4.3.1 Discussion of the Simulation Results

Matrix multiplication is benefitted the most by implementation on the array processor. Its time complexity is reduced from $O(n^3)$ to $O(n)$. The time taken to multiply two matrices is the time needed to flow both the matrices and the multiply instruction through the array. The calculations during each step are quite simple.

Matrix inversion by GE is an $O(n^3)$ operation that also can be reduced to $O(n)$ when implemented on the array processor. However, the calculations during each step of the GE algorithm require sharing results between PEs. This tends to reduce the parallelism available in the calculations. The numerical stability of the GE algorithm relies on the use of pivoting techniques. The costs associated with pivoting are discussed in the next section.

The load and unload operations have an $O(n)$ time complexity simply because of the time needed to move data through the array. These operations emphasize the high cost of communication within the array processor.

Chapter two points out that system I/O becomes the limitation to processing speed when the computational capacity of a system is increased beyond a

certain level. This is the case when the load, unload , add, subtract and scale operations are performed on a two dimensional array processor. Matrix multiplication represents a balance between I/O and computational capacity because at the maximum I/O rate of the array, all the PEs are kept busy with calculations.

4.3.2 The Cost of Partial and Total Pivoting

The partial and total pivoting algorithms described in the previous chapter are expensive in terms of the number of steps their execution requires and the complexity of their implementation. Three wavefronts are required to contain the information needed to perform pivoting:

-one wavefront for the pivoting instruction,

-one wavefront for the parameters which include the position of the element with the largest magnitude and the position of the pivot,

-one data wavefront for the value of the largest element.

The most disagreeable aspect of these two pivoting algorithms is the number of steps that they require. Partial pivoting requires the three wavefronts to travel to the southern border of the array and back up to the pivot position. The number of steps for this process is given by:

$$\textit{Number of steps for PPIV} = 2(n-p+1) \tag{4.1}$$

where p is the number of the column that is being pivoted. The value of p varies between one and $(n-1)$ during the GE process so that the total number of steps used

to perform pivoting for the entire matrix is:

$$Total\ PPIV\ steps = \sum_{p=1}^{n-1} 2(n-p+1) \tag{4.2}$$

$$= (n-1)(n+2) = n^2+n-2 \tag{4.3}$$

Similarly, total pivoting requires the three wavefronts to travel diagonally through the sub-matrix of order p and then rows and columns are interchanged as the wavefronts move the pivot back to the pivot position. Thus,

$$Number\ of\ steps\ for\ TPIV = 3(n-p+1) \tag{4.4}$$

The total number of steps to total pivot every step of GE is given by:

$$Total\ TPIV\ steps = \sum_{p=1}^{n-1} 3(n-p+1) \tag{4.5}$$

$$= \frac{3}{2}(n-1)(n+2) = \frac{3}{2}(n^2+n-2) \tag{4.6}$$

The total number of steps for each type of pivoting is compared to the number of steps required for GE of matrices of order two through eleven in table 4.2. Notice that as many steps are needed for partial pivoting of matrices of order six as for GE and that matrices of order eleven require twice as many partial pivoting steps as GE steps. Total pivoting effort for eleventh order matrices is three times the effort required for GE of the same matrix.

The pivoting algorithms exhibit a quadratic time complexity that makes their use undesirable for large matrices unless the improvement in the accuracy of

Order	GE	PPIV	TPIV
2	13	4	6
3	19	10	15
4	25	18	24
5	31	28	42
6	37	40	60
7	43	54	81
8	49	70	105
9	55	88	132
10	61	108	162
11	67	130	195

Table 4.2: Comparison of costs of GE with partial and total pivoting.

the results is worth the processing effort.

4.3.3 Comparison of the Inversion Algorithms

Stewart [14] rates LU decomposition and GE to be numerically equivalent when combined with partial pivoting. He suggests that total pivoting is used infrequently because the large number of comparisons consume too much time. These conclusions are based on single processor implementations. The implementation of these processes on an array processor changes the relative amounts of work. Table 4.3 compares the number of steps required to invert an n by n matrix on the array processor using the following algorithms:

-LU decomposition with neighbor pivoting followed by forward and back substitution (the Solve algorithm),

-GE without pivoting,

-GE with partial pivoting (PPIV),

-GE with total pivoting (TPIV).

These results are shown graphically in figure 4.8.

The implementation of LU decomposition along with the triangular system solving algorithm is much faster than GE with partial pivoting. Partial pivoting is numerically superior to neighbor pivoting, but neighbor pivoting does not add to the number of steps needed to execute LU decomposition. The need to rearrange the rows in the solution vectors because of neighbor pivoting has not

n	LU+Solve	GE	GE+PPIV	GE+TPIV
2	14	13	17	19
3	20	19	29	34
4	26	25	43	52
5	32	31	59	73
6	38	37	77	97
7	44	43	97	124
8	50	49	119	154
9	56	55	143	187
10	62	61	169	223
11	68	67	197	262

Table 4.3: Number of steps required to invert small matrices.

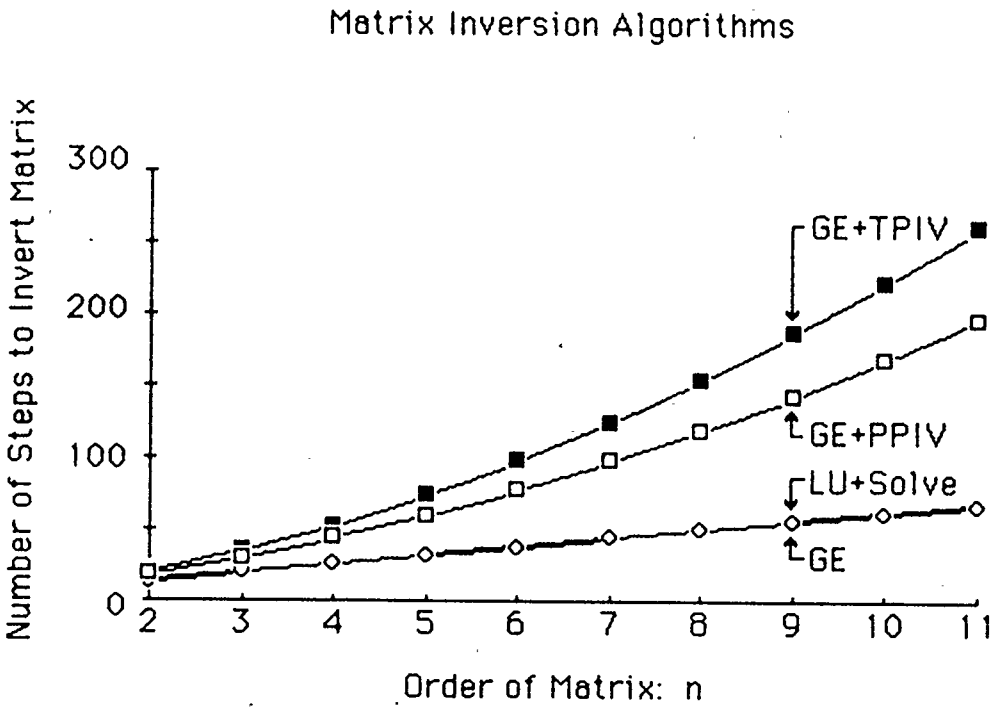


Fig. 4.8: Graphical comparison of the inversion algorithms.

been included in the performance results.

The total pivoting scheme benefits from implementation on an array processor whereas partial pivoting does not. Total pivoting requires only fifty percent more effort than partial pivoting and guarantees numerical stability.

4.4 APPLICATION OF THE ALGORITHMS TO KALMAN FILTERING

Kalman filtering is used to estimate time varying quantities from measurements that are corrupted by random errors.

The theory behind Kalman filtering is complex, however, the set of matrix equations that are repeatedly evaluated to give the optimal estimates of the time varying quantities only require the set of matrix operations given below:

-matrix addition and subtraction,

-matrix multiplication,

-matrix transposition,

-matrix inversion.

The equations for Kalman filtering [27] are described in figure 4.9.

Gelb [28] discusses the computationally intensive nature of Kalman

Prior Estimates:

$$\hat{x}_0^-, P_0^- \tag{4.7}$$

Kalman Gain:

$$K_k = P_k^- H_k^t [H_k P_k^- H_k^t + R_k]^{-1} \tag{4.8}$$

Corrector:

$$\hat{x}_k^+ = \hat{x}_k^- + K_k[z_k - H_k \hat{x}_k^-] \tag{4.9}$$

$$P_k^- = [I - K_k H_k] P_k^- \tag{4.10}$$

Predictor:

$$\hat{x}_{k+1}^- = \phi_k \hat{x}_k^+ \tag{4.11}$$

$$P_{k+1}^- = \phi_k P_{k-1}^+ \phi_k^t + Q_k \tag{4.12}$$

where:

k = discrete time index,

x = state variable vector ($nx1$),

P = error covariance matrix (nxn),

K = Kalman gain matrix (nxm),

z = measurement vector ($mx1$),

ϕ = state transition matrix (nxn),

H = matrix relationship between measurement vector and state variable vector in absence of noise (mxn),

R = covariance matrix for the measurement noise (mxm),

Q = covariance matrix for a white noise source (nxn).

Fig. 4.9: The Kalman Filtering Equations.

filtering when implemented on real time computers.

> "The burden that Kalman filtering places on real-time digital computers is considerable. Meaningful measures of this burden are storage and computation time. The first item impacts the memory requirements of the computer, whereas the second item helps to determine the rate at which measurements can be accepted. If a Kalman filter algorithm is to be programmed in real time, it is generally necessary to form some estimate of these storage and computation time requirements. These estimates can then be used to establish tradeoffs between computer size and speed and algorithm complexity. Of course, the constraints of the situation must be recognized, frequently the filter algorithm simply must be designed to 'fit'."

This last statement refers to the fact that frequently the size and the weight of the real-time computer limit the computational and memory capacity available so that approximations and assumptions must be made to simplify the computational requirements of the Kalman filter.

The Kalman filtering equations are a good application for the array processor and algorithms developed in this thesis for several reasons. Firstly, the Kalman filtering equations involve a large number of matrix operation that need to be carried out in real-time. Thus, the use of a specialized system to evaluate the equations is warranted. Secondly, the equations require the array processor to be capable of performing combinations of matrix operations. This means that intermediate results must be handled efficiently so that their I/O costs are minimized. Thirdly, some of the equations involve vectors, others involve square matrices, and some involve rectangular matrices. The array processor is flexible enough to be able to manipulate any of these shapes of matrices. Finally, the Kalman filtering application is typical of the family of high throughput, real-time applications for

which the array processor was designed.

Travassos [19] describes a systolic system designed specifically for Kalman filtering. The system uses a one dimensional systolic array for the evaluation of the predictor and corrector equations and a hexagonal array for determining the Kalman gain matrix.

The corrector and predictor equations were decoupled by Travassos to get rid of their interdependence, thus allowing them to be evaluated concurrently. The decoupled equations are given by:

$$\textit{Corrector:}\quad \hat{x}_k^+ = \hat{x}_k^- + K_k[z_k - H_k\hat{x}_k^-] \tag{4.13}$$

$$P_k^- = [I - K_k H_k]P_k^- \tag{4.14}$$

$$\textit{Predictor:}\quad \hat{x}_{k+1}^- = \phi_k\phi_{k-1}\hat{x}_{k-1}^+ \tag{4.15}$$

$$P_{k+1}^- = \phi_k\phi_{k-1}P_{k-1}^+\phi_{k-1}^t\phi_k^t \tag{4.16}$$

The Kalman gain equation remains unchanged:

$$K_k = P_k^- H_k^t[H_k P_k^- H_k^t + R_k]^{-1} \tag{4.17}$$

The article gives a stability analysis of this new set of Kalman filtering equations and considers the effect of wordlength on stability and round-off errors. The effect of sampling rate on filter sensitivity is also discussed.

4.4.1 Evaluating the Kalman Filter Equations on a Systolic Array

This section considers the sequence of matrix algorithms needed to evaluate the standard Kalman filtering equations. The decoupled equation set is

not used because it involves more calculations than the standard set. Also, the equations are to be evaluated on a single array processor, therefore, parallel evaluation of the predictor and corrector equations is not possible. Figure 4.10 lists the steps for evaluating each of the filter equations. There is certainly more than one way to evaluate these equations.

The Kalman gain equation and the corrector equation for the state variable estimates represent the majority of the array processor operations. This is partly because of the large number of matrix multiplications that involve three matrices. The matrix multiplication algorithms compute these products efficiently but the products must be reloaded into the array for further calculations. The ability to form triple matrix products with the product remaining in the array would make the evaluation of the equations more efficient.

The other three filter equations are forms that are very easily evaluated particularly due to the flexibility of Hoare's algorithm for matrix multiplication (MULT 1).

4.5 SUMMARY

Much more was learned by implementing the algorithms for the simulations than was learned from the simulation results. The performance of the algorithms is not difficult to predict and the algorithm operation can be verified by going through the algorithm by hand. However, the details of the implementations

Kalman Gain:

MULT 2	$([H_k])([P_k^-])$
MULT 1	$([H_k][P_k^-])([H_k]^t)+[R_k]$
LOAD	$([H_k][P_k^-][H_k]^t+[R_k])$
GE	$([H_k][P_k^-][H_k]^t+[R_k])$ with TPIV
MULT 2	$([P_k^-])([H_k]^t)$
MULT 1	$([P_k^-][H_k]^t)(([H_k][P_k^-][H_k]^t+[R_k])^{-1}) \rightarrow [K_k]$

Corrector:

MULT 2	$([H_k])([\hat{x}_k^-])$
LOAD	$([z_k])$
SUB	$([z_k])-([H_k][\hat{x}_k^-])$
UNLOAD	$([z_k]-[H_k][\hat{x}_k^-])$
MULT 2	$([K_k])([z_k]-[H_k][\hat{x}_k^-])$
LOAD	$([\hat{x}_k^-])$
ADD	$([\hat{x}_k^-])+([K_k])([z_k]-[H_k][\hat{x}_k^-])$
UNLOAD	$([\hat{x}_k]) \rightarrow [\hat{x}_k]$

MULT 2	$([K_k])([H_k])$
SCALE	$(-1)([K_k][H_k])$
MULT 1	$(-[K_k][H_k])+([P_k^-]) \rightarrow [P_k]$

Predictor:

MULT 2	$([\phi_k])([P_k])$
MULT 1	$([\phi_k][P_k])([\phi_k]^t)+[Q_k] \rightarrow [P_{k+1}^-]$
MULT 2	$([\phi_k])([\hat{x}_k])$
UNLOAD	$([\hat{x}_{k+1}^-] \rightarrow [\hat{x}_{k+1}^-]$

Fig. 4.10: Use of the algorithms to evaluate the Kalman filter equations.

of algorithms and the array brought to light many interesting problems. These problems included the simulation of concurrency and the passing of instructions and parameters between PEs. While these problems were difficult to solve, it was often found that the solutions provided insight into the similarities of the algorithms, the differences in PEs, and to the resources and knowledge that PE should possess.

Chapter 5

CONCLUSIONS AND RECOMMENDATIONS

5.1 CONCLUSIONS

The work described in this thesis is an examination of how a set of matrix algorithms implemented on a systolic array processor can be used to efficiently evaluate matrix equations. The need to develop algorithms that leave intermediate results in the array and that can use matrices which are already in the array was key to the achievement of this goal.

Algorithms for matrix operations previously not considered for implementation on an array processor were developed. The algorithms to load, unload, add, subtract and scale matrices were designed to follow the patterns and behavior analogous to wavefront motion. The multiplication algorithms developed by others were used separately and together to multiply matrices in a variety of situations.

An existing matrix triangularization algorithm was extended to become an algorithm for LU decomposition. Coupled with a new algorithm for solving triangular systems, it gave the array processor the ability to solve systems of equations quickly.

The Gaussian elimination algorithm with the ability to perform partial

or total pivoting allows inversion of matrices even when the matrix is susceptible to round-off errors.

All of the algorithms use the same configuration of interconnections between the PEs in the array. The execution of each of the members of this family of algorithms can be overlapped to keep all of the array processor busy almost all of the time. This overlap is possible because the flow of all of the data, parameters and instructions used in each of the algorithms adheres to the wavefront concept. The wavefront concept introduces regularity, locality, recursiveness and concurrency in the design of algorithms.

Simulation of these wavefront algorithms pointed out important details of the array processor operation and confirmed the correctness of the algorithms. The demands of the algorithms on the connectivity of the array and on the resources required by each of the processing elements were found to be modest.

Advances in the ability of systems to use parallelism and in the speed of system components, function independently to yield higher system performance. Regardless of the speed of components used to implement the array processor, the exploitation of the parallelism inherent in matrix operations through the use of parallel processing and parallel algorithms increases the number of possible real time applications.

The application of the wavefront algorithms for matrix arithmetic and

the systolic array processor to the task of evaluating the Kalman filtering equations showed that the flexibility, compatibility and performance objectives were met.

5.2 RECOMMENDATIONS FOR FURTHER RESEARCH

The search for array processor algorithms should concentrate on finding efficient ways of doing the following:

-multiplying matrices in the case where one or both of the matrices are in the array and the product is to remain in the array,

-inverting matrices using algorithms which allow PEs to perform calculations more independently and less sequentially than GE.

The lack of the former is a weak point in the set of algorithms described in this thesis. However, the cost of such an algorithm in terms of interconnections between PEs may suggest that efficient I/O management may be more practical. The existence of the latter might be found by close examination of inversion techniques that do not depend on pivoting for numerical stability.

REFERENCES

1. E. Horowitz, *Fundamentals of Data Structures,* Computer Science Press Inc, Rockville, Maryland, p. 2 (1982).

2. H.T. Kung C.E. Lieserson, "Algorithms for VLSI Processor Arrays," in *Introduction to VLSI Systems,* Addison-Wesley, Reading, Massachusetts, pp. 271-292 (1980).

3. M.A. Fischetti, "IC Designers Confront Limits to Miniaturization," IEEE *The Institute* **9** (4) p. 1 (April 1985).

4. A. Huang, "Parallel Algorithms for Optical Digital Computers," SPIE *10th International Optical Computing Conference* **422** p. 13 (1983).

5. H.T. Kung, "Why Systolic Architectures?," Internal Report #CMU-CS-81-148, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, pp. 1, 3-4, 6, 23-24 (Nov 1981).

6. E.J. Lerner, "Data-flow Architecture," IEEE *Spectrum* pp. 57-62 (April 1984).

7. J.W. Bond, "Survey of Concurrent-Processors for Signal Processing Applications," SPIE *Real-Time Signal Processing III* **241** p. 175 (1980).

8. I.E. Sutherland C.A. Mead, "Microelectronics and Computer Science," *Scientific American* **237** (9) pp. 210-220, 224 (Sept 1977).

9. A.L. Fischer H.T. Kung, "Synchronizing Large Systolic Arrays," SPIE *Real-Time Signal Processing V* **341** pp. 44-52 (1982).

10. S.Y. Kung R.J. Gal-Ezer, "Synchronous Versus Asynchronous Computation in Very Large Scale Integrated (VLSI) Array Processors," SPIE *Real-Time Signal Processing V* **341** pp. 53-63 (1982).

11. S.Y Kung K.S. Arun R.J. Gal-Ezer D.V. Bhaskar Rao, "Wavefront Array Processor: Language, Architecture, and Applications," IEEE *Trans. on Computers* **C-31,** pp. 1055-1056 (1982).

12. R.W. Priester H.J. Whitehouse K. Bromley J.B. Clary, "Problem Adaption to Systolic Arrays," SPIE *Real-Time Signal Processing V* **298** pp. 33-39 (1981).

13. C.A.R. Hoare, "Communicating Sequential Processes," *Communicatons of the ACM* **21** (8) (August 1978).

14. G.W. Stewart, *Introduction to Matrix Computations,* Academic Press, New York, pp 110-112, 131-142, 152-154 (1973).

15. H. Anton, *Elementary Linear Algebra,* John Wiley & Sons, New York, p. 5 (1984).

16. R.L Burden J.D. Faires, *Numerical Analysis,* PWS Publishers Boston, pp. 324-330 (1985).

17. W.M. Gentleman H.T. Kung, "Matrix Triangulariztion by Systolic Arrays," SPIE *Real-Time Signal Processing IV* **298** pp. 19-26 (1981).

18. H.J. Whitehouse J.M. Speiser, "Parallel Processing Algorithms and Architectures for Real-Time Signal Processing," SPIE *Real-Time Signal Processing IV* **298** (1981).

19. R.H. Travassos, "Real Time Implementation of Systolic Kalman Filters," SPIE *Real-Time Signal Processing VI* **431** pp. 97-104 (1983).

20. A.L. Fischer H.T. Kung K. Sarocky, "Experience with the CMU Programmable Systolic Chip," SPIE *Real-Time Signal Processing VII* **495** pp. 120-129 (1984).

21. J.J. Symanski, "Systolic Array Processor Implemention," SPIE *Real-Time Signal Processing IV* **298** pp. 27-32 (1981).

22. J.J Symanski, "Progress on a Systolic Processor Implementation," SPIE *Real-Time Signal Processing V* **341** pp. 2-7 (1982).

23. J.J. Symanski, "Implementation of Matrix Operations on the Two-Dimensional Systolic Array Testbed," SPIE *Real-Time Signal Processing VI* **431** pp. 136-142 (1983).

24. K.L Flanagan L.K. Woo, "The Role of Simulation in Top Down Distributed Signal Processor Design," SPIE *Real-Time Signal Processing III* **241** pp. 182-185 (1980).

25. G.M. Birtwistle, *Discrete Event Modeling On Simula,* The MacMillan Press, London, (1979).

26. B.W. Unger G.A. Lomow G.M. Birtwistle, *Simulation Software and Ada,* pp. 181-191 Simulations Councils Inc, La Jolla, Calif, (1984).

27. R.G Brown, *Introduction to Random Signal Analysis and Kalman Filtering,* John Wiley & Sons, New York, pp. 195-200 (1983).

28. A. Gelb, *Applied Optimal Estimation,* M.I.T. Press, Cambridge, Massachusetts, pp. 308-311 (1974).