APPLICATION OF A PROTOTYPE PARALLEL PROCESSING COMPUTER FOR A RECURRENT NEURAL NETWORK MODEL

by

Bekir Alper Paksoy

B.S. in M.E., Boğaziçi University, 1990

Submitted to the Institute for Graduate Studies in Science and Engineering in partial fulfillment of the requirements for the degree of Master of Science in Systems and Control Engineering

Boğaziçi University

1992

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Kemal Cılız, my thesis advisor, for his help, support, and guidance. I would like to thank Dr. Yorgo İstefanopulos for his help and guidance, Dr. Ömer Cerid for his kindness and lettting me use the Macs in his laboratory, Dr. Oğuz Tosun and Dr. Işıl Bozma for their interest in the subject, to Mehmet Göktürk for his support, and to the people who work in the Computer Center for their understanding and help. I am debted to Mr. Marshall Brain and Mr. Fred Heaton for their discussions and help about the Blitzen massively parallel processor and the Blitzen simulator. I would also like to thank to my family and my friends who granted me this thesis.

B. Alper Paksoy

ABSTRACT

In this study, Hopfield's binary neural network model is simulated using a software package simulating a bit serial single instruction multiple data (SIMD) mesh array processor, called the Blitzen massively parallel processor (BMPP). First, the parallel algorithms required for the simulation are designed. Then, the parallel implementation of the Hopfield network model has been formulated using these algorithms. The parallel implementation has been analyzed for the speed and the processor utilization. To do this, time complexity of the parallel implementation is derived and compared with the time complexity of a sequential algorithm. Finally, simulation results are analyzed and compared with the analytical derivations. The parallel algorithm described in this study for the simulation of the Hopfield network model on the BMPP architecture achieves a maximum speedup in the order of the square root of the number of processors employed. It is also shown that, for the same algorithm, the maximum possible speedup can be achieved only at a finite number of processors, and the processor utilization decreases as the number of processors is increased.

ÖZET

Bu çalışmada Hopfield'ın ikili düzen beyinsel ağ modeli Blitzen yoğun paralel işlemcisi (BYPİ) adında bit seri tek komut çok veri esaslı iki boyutlu ağ düzenindeki dizilim işlemciyi benzeten bir yazılım paketi kullanılarak benzetilmiştir. İlk olarak benzetim için gerekli algoritmalar tasarımlanmıştır. Daha sonra, bu algoritmalar kullanılarak, Hopfield ağının paralel gerçekleştirimi formüle edilmiştir. Paralel gerçekleştirim hızlanma ve işlemci yararı açısından analiz edilmiştir. Bunu yapabilmek için, paralel gerçekleştirimin zaman karmaşıklığı türetilmiş ve bu karmaşıklık sırasal bir işlemcinin zaman karmaşıklığıyla karşılaştırılmıştır. Son olarakta, benzetim sonuçları analiz edilmiş ve analitik türetmelerle karşılaştırılmıştır. Hopfield beyinsel ağ modelinin BYPİ mimarisi üzerinde benzetimi için bu çalışmada betimlenen paralel algoritma kullanılan işlemci sayısının en fazla karekökü düzeyinde bir hızlanma sağlamaktadır. Aynı algoritma için, en yüksek hızlanmanın sadece sonlu sayıda işlemciyle gerçekleştirilebildiği ve işlemci yararının işlemci sayısı arttıkça azaldığıda gösterilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

Page

Page

Page

Page

Page

LIST OF TABLES

Page

LIST OF SYMBOLS

C	The number of image classes in the Hopfield algorithm.
E_p	Efficiency achieved using P processors.
E'	Normalized efficiency ($E'=10^5*E$).
Log(x)	Logarithm of x in base 2.
N	The problem size in the Hopfield algorithm.
O(x)	A complexity function that puts an upper bound in the order of x.
P	The number of processors.
R	The number of iterations in the Hopfield algorithm.
$\Re$	The set of real numbers.
S_p	Speedup achieved using P processors.
T_p	Time complexity of a parallel algorithm using P processors.
T_s	Time complexity of a sequential algorithm
Y	The number of cycles.

I INTRODUCTION

1.1 A Historical Perspective

Today, many scientific applications benefit from the information processing capabilities of computer technology. As the number of fields using computer technology increase, these fields' dependencies and requirements also increase in various levels. These areas range from structural analysis to socioeconomics or from medical diagnosis to artificial intelligence. However, their basic demand remains the same: A feasible way of computing in reasonable time with increased reliability.

Most computer systems serve this basic demand in the same fashion as described by John von Neumann in 1940's. A control unit fetches an instruction and its operands from the memory, decodes and executes the instruction, then stores the result. To achieve the highest performance possible these tasks can be staged into multiple levels with different hardware realizations, e.g. main memory, central processing unit, I/O system realized as random access memory, cache memory, data and control bus, microprocessor, processing elements (PEs), word and bit level registers and hard disk.

Nearly all of the computers in use today are sequential. That is, they only have a single processing unit. Nevertheless, there is not a clear distinction between sequential and parallel computers because today's 'sequential' computers incorporate many parallelism properties like pipelining and memory interleaving. Actually, the basic mentality behind the parallel processing is the usage of more than one component which is the basic bottleneck in sequential processing. From this point of view if computational demands are I/O bounded, one may use multiple I/O devices or

multiple data buses or a hierarchical memory system, or if demands are processor bounded one can use multiple processors or floating point accelerators or other specially designed processors, and such systems would still be called sequential.

In its broadest meaning parallel processing is the exploitation of concurrency in the computing process using multiple devices of the same or having different devices working simultaneously or in overlapping time spans.

In an ideally sequential system, only a single component can be active at a time and all the others must wait its completion. Thus, parallel processing is a time and cost-effective solution to the parallelizable problems.

Another category is the distributed processing. Though not a precise distinction, parallel processing refers to the exploitation of concurrency by devices in a local area (at most in a room) and distributed processing refers to the information processing within geographically distributed areas. A good example to the latter is data communication networks.

There is neither a purely sequential nor a purely parallel system. The reason for the former is that today's technology allows us to incorporate parallelism properties into any computer in a cost-effective manner and the reason for the latter is that any 'single' process is sequential at somewhere through its algorithm. A sequential (or uniprocessor roughly) system can enjoy parallel properties by the multiplicity of functional units, parallelism and pipelining within the CPU, overlapped CPU and I/O operations, and the use of a hierarchical memory system [1].

For the sequential case even the personal computers of today may have many processors inside; the main processor can continue its execution while the mathematical coprocessor crunches the numbers or the I/O processor manages the storage system or the communication processor handles the communication; the main processor may be designed in components working simultaneously

to implement pipelining; the memory can be designed hierarchically to implement memory caching. A vector processing computer may have a single processor but still can perform operations on multiple numbers.

For the parallel case the solution of partial differential equations requires a computational unit communicate with its neighbors. Matrix operations demand processors to communicate with each other to find the final result. Image processing operations need a processor receive pixel values kept at neighboring processors; finally database operations need a central control unit be aware of the results after each record search. So for any single problem either a central control scheme which will synchronize and control processors or some communications between the processors are required to reach the solution. Nevertheless all the above problems can be divided into parts which are independent of each other up to a certain extent.

In 1958, Steve Unger proposed the first parallel processing computer in a paper. In 1967 a parallel working machine which was used to analyze bubble chamber tracks was implemented in University College London. Some other cornerstones were Illiac IV and Staran built in 1972, the first commercial parallel processor DAP in 1980, Massively Parallel Processor (MPP) delivered to NASA in 1983 [2], and the Connection Machine (CM) series (the latest, CM-2, finished in 1986) of the Thinking Machines Corporation [3].

There are several approaches to the realization of parallel processing computers. These are pipelining computers, vector processors, systolic arrays, array processors, and multiprocessors. A parallel computer may be classified in any of these categories [3].

Processing of images, whether binary, gray code, or color, where one or more of image pixels are assigned to a single processor, was the original and ideal application area of parallel computing This is due to the inherent parallelism in image analysis techniques like feature extraction and segmentation whose spatial structures could easily be decomposed and mapped to the processor

array. These computers, characterized by a large number (1K-64K) of processing elements (PEs), were called massively parallel computers (MPCs) or array processors. The PEs were controlled and synchronized by central control unit and were relatively simpler units with small memories (in orders of hundreds or thousands of bits), a limited instruction set, and a two dimensional mesh topology which enables PEs to communicate with their immediate neighbors.

With the advances in VLSI technology in 1980's, new interconnection topologies like n-D binary cube were developed and more applications were stimulated like information retrieval, computer vision, and artificial intelligence to benefit from the MPCs. Also the scientific computing required the processing of fixed or floating point numbers. Thus MPCs with more complex PEs (Maspar MP-1) or with a floating point accelerator assigned to a group of PEs (CM-2) were developed.

The MPCs of the present generation, both the commercial systems and the research projects, feature advanced processor and interconnection structures much more complex than the single bit mesh connected computers of the early implementations. The winning direction seems to be the combination , at some level, of different kinds of processing elements in such a way to have processing elements suited for any possible operation. Thanks to such a flexibility, MPCs can be considered the most effective parallel computer solution in many of the mentioned application fields [2].

1.2 Scope, Purpose, and Outline of This Thesis

Inspired from the nervous system, neural nets display behavioural similarities with human beings while learning. However, artificial simulation of nets suffers from the large

computational burden imposed by the iterating training sessions. A training session comprises matrix and vector operations with elements going through nonlinear functions. Especially multilayered nets require a large number of training sessions because of the diminishing effect of error correcting terms on connection weights while backpropagating.

Matrix operations of inner product, matrix-vector product, and matrix-matrix product have computational complexities in orders of N, N^2, and N^3, respectively. The sequential machines directly reflect the computational complexity of an algorithm to the time complexity of the algorithm. Though parallel computing leaves NP problems as they are, it offers, for example, speedups in the orders of $N^3/\log N$ in matrix multiplications. With the recent decrease in hardware prices, MPCs (Massively Parallel Computers) can be used more effectively as means to decrease the time complexity of the computational algorithms.

In this study, we will try to simulate an artificial neural network model, called the Hopfield network model, using a software simulator that simulates a massively parallel processor, called the Blitzen massively parallel processor (BMPP).

We will first design the parallel algorithms required for the simulation. Then, we will formulate the parallel implementation of the Hopfield network model using these algorithms. The parallel implementation will also be analyzed for the speed and processor utilization. To do this, time complexity of the parallel implementation will be derived and compared with the time complexity of a sequential algorithm. Then, we will simulate the Hopfield network model using code written in the Blitzen simulator's language. Finally, the simulation results will be analyzed and compared with the analytical derivations.

II BASICS IN PARALLEL PROCESSING

2.1 Parallel Computing Classes

Parallel computing has developed according to the computational needs of applications. The differences between the parallel computing classes are largely based on these needs.

The first class of needs are exemplified by the matrix operations, finite difference, and finite element methods of PDE solutions used in weather forecasting, oceanography and astrophysics, image processing, reservoir modelling, and plasma fusion power studies. The FFT of signal processing, the real-time or off-line searching, matching, merging, and sorting operations on data bases used in documentation retrieval, analysis of text and memory based reasoning (linguistic pattern recognition), air traffic control also have similar needs. Some other elements of the first class are the simulation studies in computer vision, object recognition, neural nets, molecular dynamics, VLSI design, and computer assisted tomography [1,3,4,5].

In the above problems the data processed or the variables manipulated can be divided into multiple streams such that each computational unit handles a stream. Such problems are called *data* or *variable parallel.* They can be decomposed along a spatial dimension; a different computational unit can be assigned the task of manipulating the data or variables associated with a small region in space. A system choice for the data parallel problems among the existing systems may lead to a uniprocessor with multiple functional units (a pipelining computer or a vector processor) running in a single-instruction single-data (SISD) fashion. A better choice can be a system comprising a large number of simple processors working together in a single-instruction multiple-data

(SIMD) fashion (systolic arrays or array processors) since the interactions between the processors are local in most of the above problems.

Another class deals with the computation, analysis, simulation and optimization of large scale systems, general system of equations and mathematical programming. These problems can be divided into a smaller number of subtasks, and these subtasks appear to be more complex than the subtasks of a highly data parallelizable problem. Such problems are also called *instruction parallel,* and a system with fewer but more powerful processors under a more complex control mechanism is needed. A multiprocessor system with semi-autonomous and loosely connected processors is a good candidate for such systems [4].

A third class is information acquisition, extraction, and control within geographically distributed systems. This class, named as distributed processing, can be considered as a special case of parallel processing with a loose or no synchronization. Contrary to the other forms of parallel processing, communication and reliability have the utmost importance in distributed processing. Two examples are the sensor and the data communication networks [4].

The above needs lead to the discrimination of parallel processors in the respects of type and number of processors (simple vs. complex and coarse grained vs. fine grained), presence or absence of a central control mechanism (single instruction or multiple instruction), synchronous vs. asynchronous (local vs. distributed processing), and processor interconnection topologies [4].

Only the most popular two of these classification schemes are considered in this study. The first classification is based on the multiplicity of instruction and data streams a computer executes [1]. A sequential computer executes a single instruction stream on a single data stream. A parallel computer can execute single/multiple instruction stream(s) on single/multiple data streams.

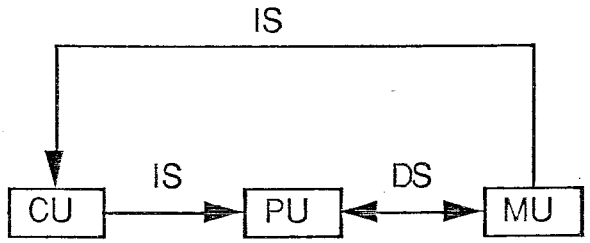
The second classification is loosely based on the combined structural properties of a system, e.g., number of processors in the system, tightness of coupling between these processors, how specialized these processors are, etc. The degree of parallelism in the sense of autonomy of individual processors is taken as the degree of complexity. Therefore, a pipelining system is taken as the simplest parallel system and a multiprocessing system is taken as the most complex parallel system because the latter also deals with the problems of the first one, and it also has complex synchronization problems. In this study, we will focus our attention to systems with multiple processors which execute the same instruction simultaneously.

2.1.1 Architecture Based Classification

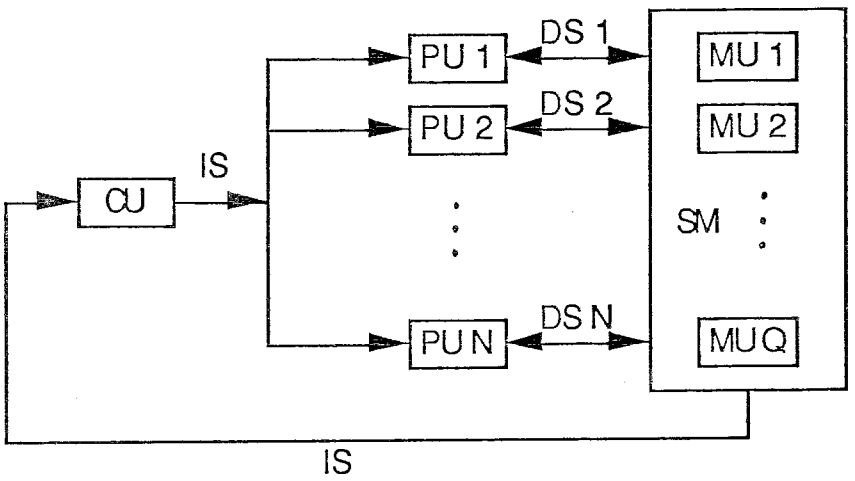
Any computer must handle data or instructions in a single stream or in multiple streams. The computing model thus chosen determines the architecture of a computer.

The classification of computers based on the multiplicity of streams was first proposed by Michael J. Flynn in 1966. Flynn has used three basic conceptual components in his system models: memory units which keep instructions and data, a processing unit which executes instruction on data, and a control unit which decodes and sends the instructions to the processing unit. Each instruction stream is generated by a single control unit. Data streams flow between processors and memory modules bidirectionally. Computers can be classified into four groups based on the four models given below [1].

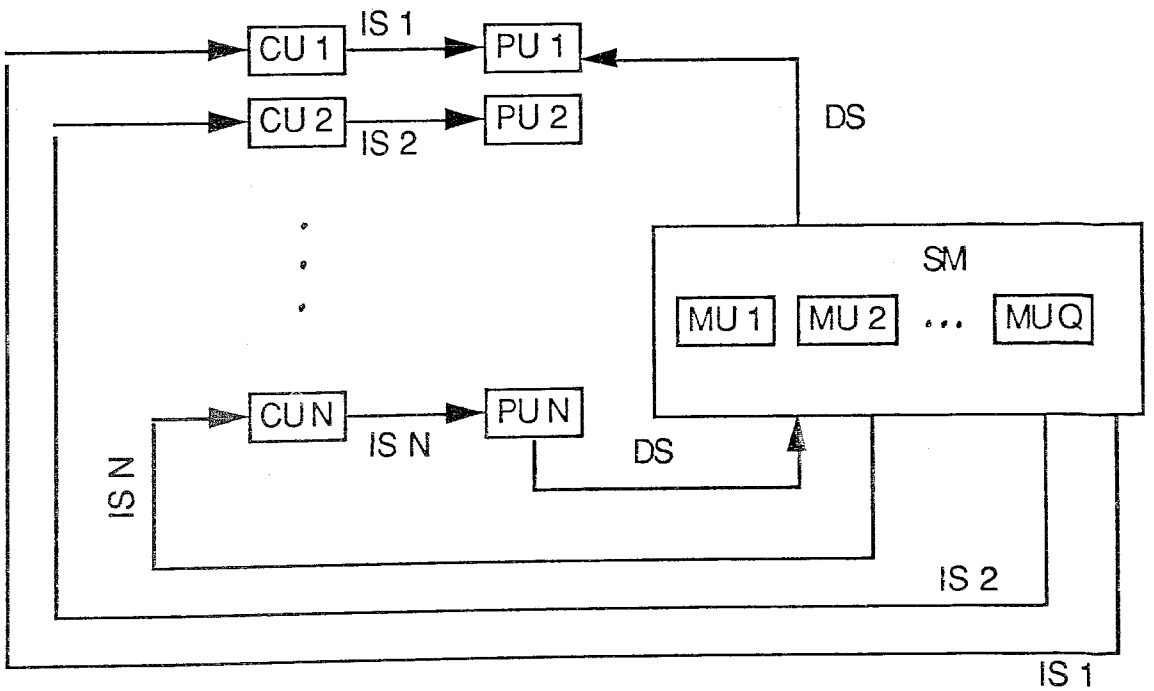
Single Instruction Single Data (SISD) Model: This model has only one of each of the basic system components. It is the basic model for the sequential computers. Most SISD systems have pipelining processors. This model was actually proposed by John von Neumann in 1940's. Fig. 2.1 (a) illustrates the SISD model



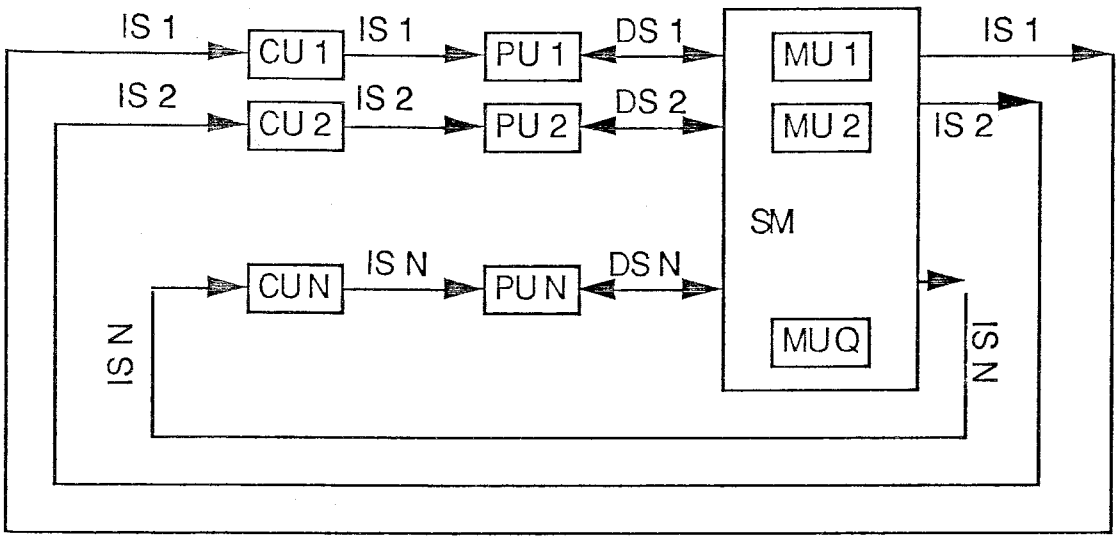
(a) The SISD model.



(b) The SIMD model.



(c) The MISD model.



(d) The MIMD model.

Fig. 2.1 The computational models [1]. CU: control unit, PU: processor unit, MU: Memory unit, SM: shared memory, IS: instruction stream, DS: data stream.

Single Instruction Multiple Data (SIMD) Model: This model is the basic model for array processors. Each processor receives the same instruction from the control unit and executes this instruction on its local data. Processors in a SIMD model are the simplest type of processors in all models. They communicate either through a shared memory or an interconnection network. Fig. 2.1 (b) illustrates the SIMD model.

Multiple Instruction Single Data (MISD) Model: This model nearly has no applications and has been considered as fruitless. But for the sake of completeness it can be mentioned that multiple control units sends decoded instructions to the processors for them to execute these on the same data. Note that, processors of such a system would be more complex than of an SIMD model. Fig. 2.1 (c) illustrates the MISD model.

Multiple Instruction Multiple Data (MIMD) Model: In general, multiprocessor and multicomputer systems are based on this model. Each processor in such a system executes the instruction coming from its control unit on its local data. The processors of such a system are preferred to have certain features to support process

recoverability, efficient context switching, large virtual and physical address spaces, effective synchronization primitives, an interprocessor communication mechanism and an adequate instruction set [1]. Note that if processors do not share a common memory, they would rather be called multiple SISD (MSISD) computers [1]. Fig. 2.1 (d) illustrates the MIMD model.

2.1.2 Structure Based Classification

Pipelining Computers: In general pipelining is a time-effective solution to realize temporal parallelism in digital computers. Similar to the assembly lines in an industrial plant, tasks are divided into a sequence of subtasks which are carried out by different functional units in overlapping time spans. Pipelining computers have overlapping data processing capabilities in the central processor, in the I/O processor, or in the memory hierarchy [1].

Linear pipelining is the design of a pipeline in cascaded processing stages. These stages perform arithmetic and logic operations over the data stream flowing through the pipe. They are separated by high speed interface latches which are fast registers for holding the intermediate results between the stages [1]. Instruction, arithmetic and processor pipelining are common approaches in the processor pipelining [1]. Fig. 2.2 depicts the instruction pipelining.

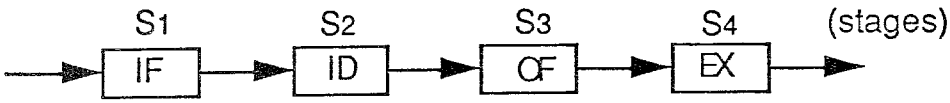


Fig. 2.2 The instruction pipelining. IF: instruction fetch, ID: instruction decode, OF: operand fetch, EX: instruction execute.

Vector Processing Computers: A vector is an ordered set of elements where an element can be a floating point or a fixed point number, a logical quantity, or a byte. A vector operation has, at least, a vector operand and optionally a scalar or another vector operand. It may

yield either a vector or a scalar. Vector processors are candidates of fast vector/matrix operation performers and are based on high performance floating point processors and vector oriented memory organizations. Pipelining is common to all modern vector processors also [2].

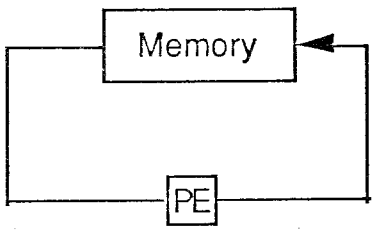
The power of vector processing computers stem from their abilities to manipulate all the variables of a vector operand at a single step. Table 2.1 shows operations which can be found in modern vector processors [1].

Description	Operation
Vector square root	$B(j) = \sqrt{A(j)}$
Vector sine	$B(j) = \sin(A(j))$
Vector summation	$S = \Sigma A(j)$
Vector maximum	$S = \max(A(j))$
Vector add	$C(j) = A(j) + B(j)$
Vector multiply	$C(j) = A(j) * B(j)$
Vector larger	$C(j) = \max(A(j),B(j))$
Vector-scalar add	$B(j) = S + A(j)$
Vector-scalar divide	$B(j) = A(j)/S$

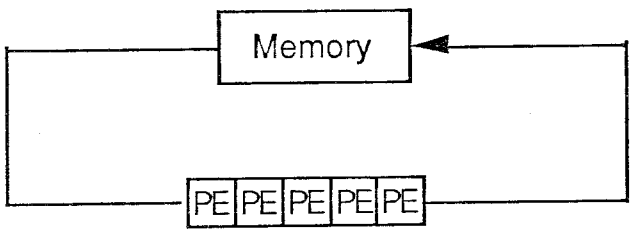
Table 2.1 Vector operations. A,B, and C are N-tupled vectors, and S is a scalar.

The first two examples of vector processors, CDC-Star-100 and TI-ASC, appeared in the 1960's. Cray-1 and Cyber-200 (1970's) series were the second generation machines. Cray X-MP (1983) had dual processors. The next generation of vector processing computers like Cyber-2xx and Cray-2 have very powerful multiple processors [1].

Systolic Arrays: As the hardware costs decline, it has become more feasible to implement parallel algorithms that realize computationally bounded algorithms directly in the hardware using the rapidly advancing VLSI technology. Examples to these computationally bounded algorithms are FFT, L-U decomposition, matrix multiplication, and feature extraction algorithms [1].



(a) A conventional processor



(b) A systolic array.

Fig. 2.3 The concept of systolic array. PE: processing element.

A systolic array can exploit a wide class of computationally bounded algorithms where multiple operations are performed on each data item in a repetitive manner. Once a data item is brought out from the memory it can be used effectively at each cell it passes. Thus, information in a systolic system flows between the cells in a pipelined fashion, and communication with the outside world occurs only at the boundary cells. The memory fetches act similar to the pulses of the heart that pushes data in and out of the systolic array. Fig. 2.3 displays a conventional processor vs. a systolic array.

Array Processors (Massively Parallel Computers): A sequential computer's speed can be improved in various ways. Unfortunately, since these improvements must combine the latest technological innovations, such designs tend to be very costly. Array processors offer a feasible way of achieving the desired performance.

Array processors were first proposed in 1958 [2]. Their most distinguishing feature is the huge amount of processing elements they possess in a repetitive structure. This architecture is called *fine grained parallelism* contrary to the architecture of a multiprocessor system, called *coarse grained parallelism* with fewer but more complex processors. Table 2.2 lists estimated speeds for three different massively parallel computers (MPC's).

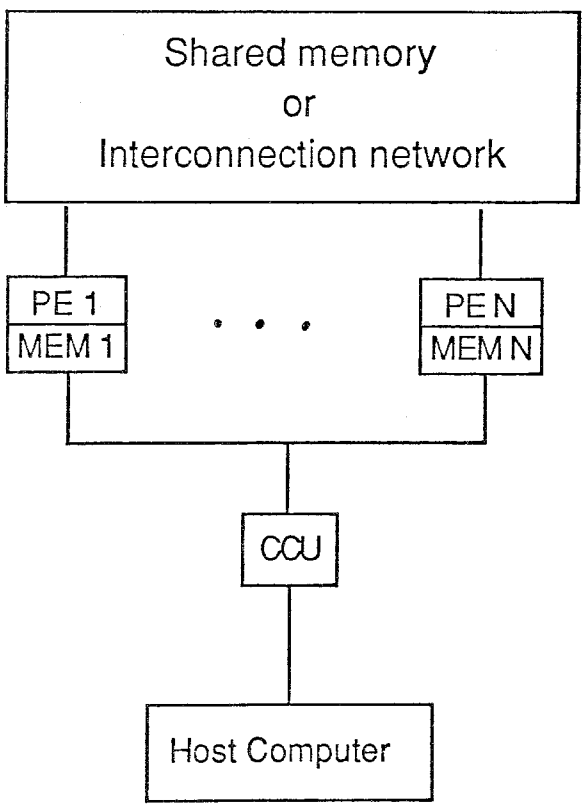


Fig. 2.4 A generalized array processor. A host computer provides the user interface, usually, by a high level programming language. It detects and sends the parallel parts of the code to the central control unit (CCU). The CCU decodes and broadcasts these instructions to the PEs. Every PE has its local memory (MEM) and communicates with each other either by a shared memory or an interconnection network.

Another characteristic of array processors is the way they receive data and execute instruction streams. Though it is theoretically possible that an array processor can be organized in MISD or MIMD fashion, the latter has rarely been implemented, and the former has never been implemented. SIMD is the most suitable architectural configuration because of the inherent data parallelism in most of the problems. Instruction parallelism is hard to achieve because of the highly complex algorithm threading and synchronization issues. The array structure is the most efficient for algorithms which makes every PE execute the same instruction. However, a partitioned MIMD/SIMD approach is also possible as exampled in Connection Machine and Image Understanding Array [6,3]. Fig. 2.4 shows the general layout of the functional parts of an array processor.

Operation	MPP	CM	DAP
Addition			
8-bit fixed	6553	4000	-
32-bit fixed	3343	3300	-
32-bit floating	470	4000	-
Multiplication			
8-bit fixed	1861	-	1600
32-bit floating	291	4000	-

Table 2.2 Processing rates in millions of operations per second (MOPS) for different parallel processing systems.

PEs of an array processor are often simple computational units, usually with one bit registers, i.e., serial, and a limited instruction set that may contain no more than a few logical operators. They are simple arithmetic logic units without any instruction decoding capability. Usually, addition is the only mathematical operator, and some other instructions to accomplish communication between the PEs or between the PEs and the main memory are also available. They are also called data processors because they execute microlevel instructions broadcast from the

central control unit (CCU, defined in Fig. 2.4) on their local data [3]. However, bit serial processors are quite slow for word-length operations. Some designers have preferred assigning a floating point accelerator (CM-2) or a coprocessor (Digital Array Processor (DAP)) to a cluster of PEs or using PEs with internal registers of higher bit lengths and internal floating point accelerators (The MasPar family) [2].

In massively parallel processors, PEs communicate with each other via either an interconnection network or a shared memory. Many interconnection topologies ranging from one dimensional to n dimensional networks has been developed. The problem here is to choose the most suitable topology for a specific problem. If the shared memory approach is preferred, one must choose a memory access model to meet the PE memory access demands. Various interconnection schemes for interconnection networks and four memory access models for shared memory communications will be discussed later in this section.

Another class of array processors is *associative processors.* Associative processors use associative memories (AM) which are *content addressable* (data accessed by content instead of address contrary to the random-access memory (RAM)). Thus, parallel access to multiple words in memory are allowed. Associative processors are used as text retrieval computers and back-end database machines [1]. Some examples of associative processors are the Goodyear Aerospace STARAN, the Parallel Element Processing Ensemble [1], and the Airborne Associative Processor [7].

The first working processor array, completed at University College London in 1967, had a two dimensional mesh topology with 20 by 20 PEs. It was used to analyze bubble-chamber tracks [2]. Illiac IV with 64 PEs interconnected under a two dimensional mesh was fabricated by Burroughs Corporation and delivered to NASA in 1972. Illiac IV was designed to perform matrix and vector computations [1]. Burroughs Scientific Processor (BSP), by Burroughs Corporation (1979), used a crossbar interconnection network to connect PEs and memory modules (shared memory configuration)

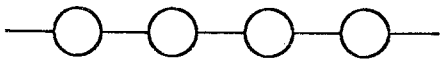
[1]. The processors of the Digital Array Processor, (1981) by the International Computer Limited in England, could be grouped into array sizes ranging from 16 by 16 to 256 by 256. The Massively Parallel Processor (MPP) of Goodyear Aerospace delivered to NASA in 1983 [2] had 16K processors arranged on a two dimensional mesh topology. The MPP was designed for high speed processing of satellite imagery [7]. CM-1 in 1985 and CM-2 in 1987 by the Thinking Machines Corporation provided multiple topologies. CM series were intended for general purpose computing. Some other current array processors are the AIS family from Applied Intelligence Corporation, and the MasPar family from MasPar Computer Corporation [2]. Table 2.3 summarizes the developments in array processors chronologically.

Computer	Type	Year
Unger	network	1958 (proposed)
Illiac-IV	network	1972
Staran	associative	1975
BSP	shared memory	1979
DAP	network	1981
MPP	network	1983
CM-1	network	1985
CM-2	network	1987

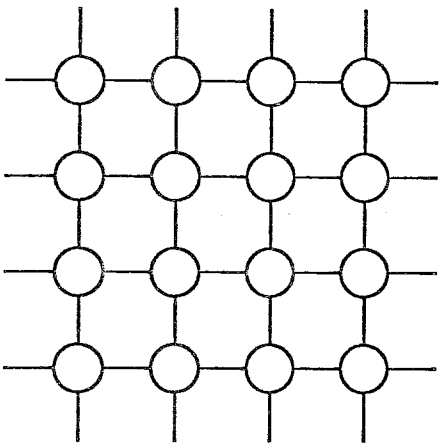
Table 2.3 Developments in array processors.

In the following, we describe in more detail the two basic schemes for the interprocessor communications in array processors,

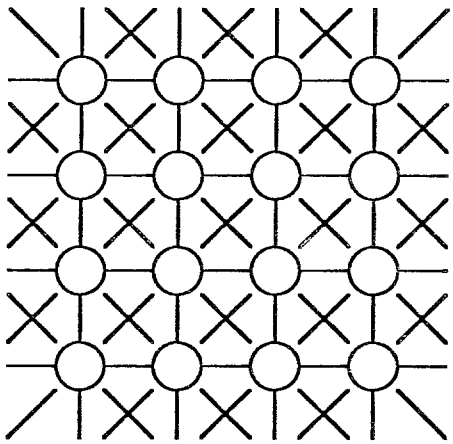
Interconnection Network Communications: In most of the massively parallel computers, PEs communicate with each other by passing data items to each other through links provided by the interconnection network.



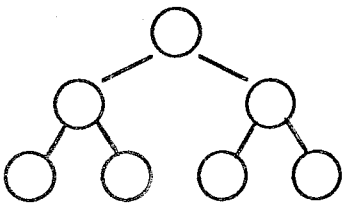
(a) Linear array.



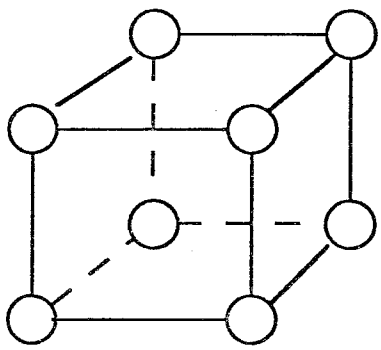
(b) 2-D mesh.



(c) X-grid.



(d) Tree.



(e) 3-d cube.

Fig. 2.5 PE interconnection schemes.

The most desirable intercommunication scheme is a fully connected one but it is very costly. Thus, instead of a fully connected scheme, other schemes have been proposed and implemented.

For image processing problems the most popular scheme is the two dimensional mesh since the spatial structure of numerous image processing techniques can be decomposed into a two dimensional array. Some of the other schemes are linear array, X-grid, tree connection, and n-d cube (3-d cube given in Fig. 2.5). These network topologies are illustrated in Fig. 2.5.

Most of the MPC's provide a single interconnection scheme. However, some recent MPC's like the CM series allow multiple interconnection schemes. CM-1 has a mesh topology but also can switch to a tree and a relational graph. CM-2 has a hypercube topology and can switch to a mesh topology [3].

The topology of an MPC is highly important since the communication complexity may become the governing term in the time complexity of a parallel algorithm. This subject will be further discussed in the following sections.

Shared Memory Communications: PEs communicate through a common memory in this configuration. When a PE wants to pass a data item to another PE, it writes this data item to a memory location also accessible by the other PE. Therefore the local memories of PEs become common, i.e,. shared.

There are four models of memory access: exclusive-read and exclusive write (EREW), concurrent-read and exclusive-write (CREW), exclusive-read and conclusive-write (ERCW), and concurrent-read and concurrent-write (CRCW) [8]. Concurrency implies that an operation can be done on a memory location by more than one processor simultaneously; exclusiveness implies that only one processor can accomplish an operation on a memory location at any given time.

While CRCW is the most general model, it is impossible to implement it completely because of apparent write conflicts. Instead, concurrency in writing can be achieved by writing into the memory a certain result derived (like the addition of all or a logical operation between them or the maximum or minimum) from the data items sent by the PEs.

Unfortunately the shared memory configuration is very costly to realize since when one processor needs to gain access to a datum in memory, some circuitry is needed to create a path from that processor to the location in memory holding that datum. The cost of such circuitry is usually expressed as the number of logical gates required to decode the address provided by the processor [8]. For an M locations memory with N processors cost of circuitry is proportional to N*f(M). This difficulty can be overcome by dividing memory into modules and using an interconnection network between the PEs and the memory modules. Nevertheless, such realizations leads to models even weaker than EREW. As an example to this, Burroughs Scientific Processor (BSP) uses a crossbar switch network as the intercommunication network.

Multiprocessor Computers: The instruction stream of an *instruction parallel* problem can be divided into multiple streams. Since these streams are usually few and complex, instruction parallel problems can not efficiently utilize fine grained parallelism of array processors.

Processors in a multiprocessing system cooperate semiautonomously with a loose communication between them. They run simultaneously under a single operating system. Multiprocessing systems with pipelining processors, or with pipelining and vector crunching processors, are very suitable for instruction parallel problems.

Multiprocessor systems are characterized by common sets of memory modules, shared peripheral devices, and I/O channels [1]. However, each processor will have its own local memory memory

and devices which are completely allocated to it. Multiprocessor systems utilize the MIMD architecture in large.

The interprocessor communication schemes are time shared common bus, crossbar switch network, and multiport memories [1]. The selection and design of components of a multiprocessor system highly depends on its being a loosely or tightly coupled system. Note that a multiprocessor system still refers to a centralized processing scheme whereas a multicomputer system refers to a distributed processing scheme which comprises several autonomous computers.

Some of the commercially available multiprocessor systems are IBM 370 and 3080 series, Univac 1100 series, the Tandem Nonstop system, the HEP, the Cray X-MP, and the Cray-2 [1].

2.2 Parallel Algorithms

2.2.1 Design Approaches

Two approaches can be followed to design a parallel algorithm (All the discussions below are about the array processor parallel algorithms). These are:

1) To use an existing technique as it is. This approach can be divided into two different forms.

   a. To use a sequential algorithm for that technique and, to code as if the program will run on a sequential computer and let the compiler detect parallelizable parts of the code.

b. To design a parallel algorithm for that technique and to code using a programming language providing special instructions for the utilization of the parallel processor.

2) To invent a new technique or to design a new algorithm utilizing a parallel processor heavily.

Usually, the first approach is followed, since designing a new technique will not be easy, and it is doubtful that a new design will be more efficient than the existing one.

Before giving examples for the above approaches, a hypothetical array processor and its pseudo programming language are described.

A Hypothetical Array Processor: The hypothetical array processor (HAP) has an architecture very similar to the array processor used in the software simulations of this study (See Appendix B for more information on the array processor used in this study). The HAP is an SIMD mesh array processor with P processing elements (PEs). All the PEs are controlled by a central control unit. A local RAM acts as a cache memory for the PEs, but being on the same chip with the processors, it is small in size. A large random access memory, called video RAM (VRAM), acts as a buffer between the local memory and the main memory. PEs can not communicate via VRAM because VRAM area accessible by each PE is different.

A PE can accomplish basic logical and mathematical operations. It has certain instructions to move data between its registers, its local memory, and the VRAM. It can send and receive data items from its immediate neighbours in north, east, west, south (NEWS), and four other diagonal directions simultaneously. (These four additional links along with NEWS links is called an X-grid. See Fig. 2.5 in Section 2.1.2 for an X-grid).

The HAP has a pseudo-language similar to the C programming language but also has instructions special to the array processor architecture. When such an instruction is detected during the execution, it is sent to the central control unit by the host computer. Below are some of the instructions from the HAP's pseudo-language.

SendToDirection1ReceiveFromDirection2(x,y): This is an instruction to receive a data item in variable x from the PE in direction Direction1 and to send a data item in variable y to the PE in the opposite direction to the direction Direction1, namely Direction2. Send and receive operations are simultaneous; therefore, if x and y are the same as x, x keeps the received value while the old value of x is sent.

AND: This instruction bitwise ANDs its operands.

OR: This instruction bitwise ORes its operands.

SetRoute(x): This instruction enables a wraparound or grid mode of routing; x can be "torus" or "grid". See Section 3.1.3 for an explanation of routing modes.

Other features of the HAP and its instruction set will be further explained in the subsequent sections if and when necessary.

A noise removal algorithm that removes white pixels from a dark background is a good example to the first of the parallel algorithm design approaches. This algorithm turns a pixel's value to black if all of its immediate NEWS neighbours are black. There are P pixels. The sequential code would be as in Fig. 2.6. In Fig. 2.6, old(i,j) holds the original value of pixel (i,j) and new(i,j) holds the computed value of pixel (i,j).

```
for (i=1 to √P){
    for  (j=1 to √P){
        new(i,j)=(old(i-1,j)  and  old(i+1,j)  and  old(i,j-1)
          and old(i,j+1)) or old(i,j);
    }
}
```

Fig. 2.6 A sequential noise removal algorithm.

A parallelizing compiler may detect the do loops and produce the executable file accordingly. The same algorithm can be written using the pseudo-parallel language of the HAP as an example to part (b) of the first approach. The parallel algorithm is given in Fig. 2.7

```
FOR ALL PEs (i,j)
    SendToSouthReceiveFromNorth(old,north);
    SendToWestReceiveFromEast(old,east);
    SendToEastReceiveFromWest(old,west);
    SendToNorthReceiveFromSouth(old,south);
    new=(north and east and west and south)
      or old;
```

Fig. 2.7 A parallel noise removal algorithm.

In Fig. 2.7, the variable *old* holds the original value of a pixel, and the variable *new* holds the computed value of a pixel. An example to the second approach can be the utilization of a new technique which can exploit the fine grain parallelism of array processors for the specific problem. In fact, such approaches are being developed [9].

There are three important points that must be considered while designing or realizing a parallel algorithm. The first is the *computation cost* measured, as the number of basic mathematical or logical operations performed; the second is is the *number of processors* and the *network topology,* and the third is the *communication cost* measured as the number of routing operations performed.

The *time complexity,* which is expressed in units of time, of a parallel algorithm is the addition of the times spent to pay these costs. Note that the communication cost does not exist in sequential algorithms, but can be the dominant term in the time complexity of a parallel algorithm. Therefore, the above mentioned points should be examined carefully for any algorithm.

The ultimate goal in designing a parallel algorithm is to get shorter solution times, hence, speedup with respect to the sequential implementations. To be able to achieve the largest speedup, one needs a methodology to design parallel algorithms. However, formal models are more than needed for the purposes of this study. The next two sections briefly present a method named directed acyclic graphs (DAGs) and how DAGs can be used to represent, design, develope, and analyze parallel algorithms.

At this point an acknowledgement is in place. The book "Parallel and Distributed Algorithms" by D. P. Bertsetekas and J. N. Tsitsiklis is one of the excellent books in its subject. The Sections 2.2.2 to 2.2.5 are largely based on the 1st chapter of this book.

2.2.2 Directed Acyclic Graphs

A *directed graph* (see Appendix B of [4] for more information about graphs) is a finite non-empty set N of nodes n_i and a collection A of ordered pairs of distinct nodes from this set. Each ordered pair of nodes (n_i,n_j) in A is called an *arc*. There is only a single arc between a pair of nodes, but this arc can either be unidirectional or bidirectional.

A *path* P in a directed graph is a sequence of nodes $n_1,..,n_k$ for $k \geq 2$ and a corresponding sequence of k-1 arcs such that the i^{th} arc in the sequence is either (n_i,n_{i+1}) (a *forward* arc) or (n_{i+1},n_i) (a *backward* arc). Nodes n_1 and n_k are denoted as *start* and *end nodes* of P, respectively. A *cycle* is a path for which the start and the end nodes are the same. A *positive cycle* is a cycle for which all the arcs are forward.

A directed graph with no positive cycles is a *directed acyclic graph* (DAG) (see [4] for more information about DAGs). Thus, any path in a DAG graph can pass through a node once, and all the arcs in a path are both unidirectional and in the same direction.

A DAG consists of also a set N of nodes $\{n_1,...,n_N\}$ and a set A of directed arcs (n_i,n_j). Each node n_i represents an operation and each arc represents a data dependency. An operation can be a boolean operation, an arithmetic operation, or a complicated operation like the execution of a subroutine. As an example, a DAG that represents $(a+b)^2$ is given in Fig. 2.8.

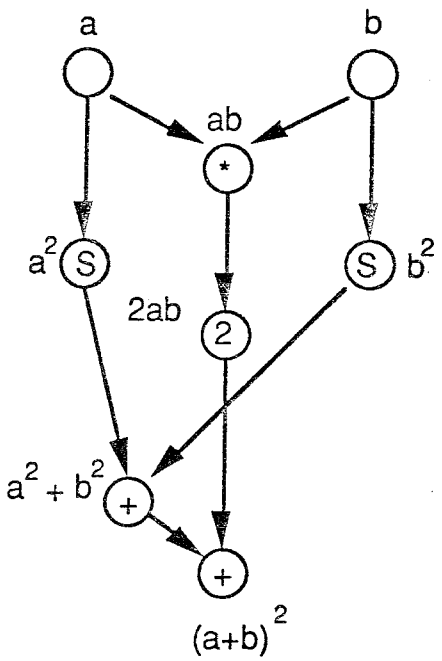


Fig. 2.8 A DAG for $(a+b)^2$.

In a DAG, if $\{n_i,n_j\} \in N_o$ and $(n_i,n_j) \in A$, then n_i is said to be a *predecessor* of n_j. The *in-degree* of node n_i is defined as the number of its predecessors and *out-degree* of node n_i is defined as the number of processors for which node n_i is a predecessor. Thus, an *input node* has an in-degree of zero and, an *output node* has an out-degree of zero. N_o is the set of nodes except the input nodes. A *positive path* is a sequence $n_0,..,n_K$ of nodes where $(n_i,n_{i+1}) \in A$ for i=0,..,K-1. K is called the *length* of a path. The *depth* D of a DAG is the maximum of the lengths of all the positive paths. D is positive in a DAG because of acyclicity and the positive path yielding D must obviously start at an input node and end at an output node. Note that the depth of the DAG in Fig. 2.8 is 3.

Now, assume that we have a pool of processors. Let each node n_i is assigned a processor P_i except the input nodes and, each processor is capable of performing the operation corresponding to that node in unit time. We further assume that communication between the processors is instantaneous.

To be able to represent an algorithm fully by a DAG, one also needs to fix when an operation is performed by which processor. This can be done via a *schedule* .

There are two constraints that must be imposed on a schedule.

1. A processor can perform at most one operation at a time. Therefore, if $n_i,n_j \in N_o$ for $i \neq j$ and $t_i = t_j$, then $P_i \neq P_j$ where t_i denotes the time at which the i^{th} step of a schedule ends. Note that more than one processor can be active at a single step.

2. If $(n_i,n_j) \in A$, then $t_j = t_i+1$ (each operation takes unit time).

Since input nodes are assigned no processors, their completion times, t_i, are zero.

After fixing P_i and t_i, a DAG is said to be scheduled, and the set $\{(n_i,P_i,t_i) \mid n_i \in N_o\}$ is called a schedule S.

2.2.3 Computation Complexity

The computation complexity is the time paid for the computation cost of an algorithm. The following notation will be used throughout the text:

A is a subset of real numbers $\Re$. For f: $A \rightarrow \Re$, g: $A \rightarrow \Re$,

1. $f(x) = O(g(x))$ implies, for any $x \geq x_0$, $f(x) \leq cg(x)$,

2. $f(x) = \Omega(g(x))$ implies, for any $x \geq x_0$, $f(x) \geq cg(x)$,

3. $f(x) = \Phi(g(x))$ implies, for any $x \geq x_0$,

$$c_1 g(x) \geq f(x) \geq c_2 g(x),$$

4. log(x) is logarithm base 2 of x,

where $x_0, c, c_1, c_2 \in \Re^+$.

In the above notation, O(.) puts an upper bound in the order of its argument, Ω(.) puts a lower bound in the order of its argument, and Φ(.) denotes the order of its argument.

For a DAG *G* and a schedule *S*, $\{(n_i,P_i,t_i) \mid n_i \in N_o\}$, of P processors, the time spent is $\max(t_i)$. The time complexity, T_p, of an algorithm is defined as the minimum of the times spent by any schedule S that realizes the same algorithm described by G using P processors.

Obviously, a schedule using infinitely many processors yields the minimum time complexity. However, note that for some $P=P^*$, where P^* is a positive finite integer, $T_\infty = T_{P^*}$ (At least think of the case with one processor per each node in N_0). Thus, the minimum

time complexity can be achieved with a finite number of processors. Therefore, T_∞ can be defined as

$$T_\infty = \min_{p \geq 1} T_p \qquad (2.1)$$

Further note that T_1 is the time complexity of a G and S pair simulated by a single processor and is equal to $|N_0|$. Therefore,

$$T_1 \geq T_p \geq T_\infty \qquad (2.2)$$

Another important property of T_∞ is its equality to the depth D of its graph G. This is because a schedule can not be completed without traversing all the positive paths, and no positive path can be longer than the path with a depth D.

Now, let's state some properties of T_p:

1. Assume that there is a single output node and in-degree of each node is at most two. Therefore, for n input nodes

$$T_\infty \geq \log(n).$$

2. If $q = c.p$, then $T_p \leq c.T_q$.

3. If $p = \Omega(\frac{T_1}{T_\infty})$, then $T_p = O(T_\infty)$.

4. If $p = O(\frac{T_1}{T_\infty})$, then $T_p = \underline{\Theta}(\frac{T_1}{p})$.

These facts are of fundamental importance. The first puts a lower bound on the time complexity of any algorithm independent of the number of processors. The second states if the number of processors is increased by c, then the speedup would be less than c. The third states if the number of processors are bounded below in the order of T_1/T_∞, then the speedup is bounded above in the order

of T_1/T_∞. The fourth states if the number of processors are bounded above in the order of T_1/T_∞, then the speedup is in the order of T_1/T_∞.

The above discussions show us that we can reach T_∞ or T_∞ within a constant factor using a finite number of processors. Thus, it is tactful to develop a DAG and a schedule as if we have infinitely many processors, and then, adapt the algorithm to the available number of processors. Properties three and four also imply that the T_1/T_∞ is a limit point for the processor utilization. Indeed, one might utilize the processors best by choosing P equal to T_1/T_∞.

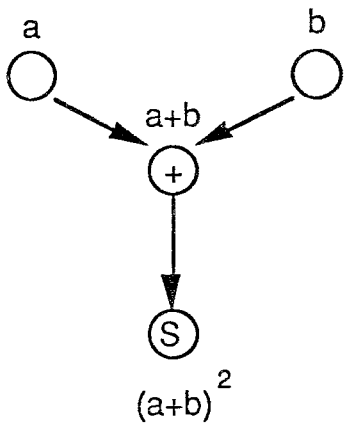


Fig. 2.9 Another DAG for $(a+b)^2$.

Note that all of the above discussions are about the time complexity of a G and S pair but not the time complexity of the algorithm itself because the time complexity of an algorithm (or rather a technique) is implementation dependent. So anyone designing a parallel algorithm must choose a spatially decomposable technique and must be careful about the optimization of his/her implementation for an optimal time solution. Fig. 2.9 displays this fact by giving another DAG for $(a+b)^2$ with a smaller depth.

2.2.4 Communication Complexity

The communication complexity is the time paid for the communication cost of an algorithm. It is as important as the computation complexity since it may contribute to the time complexity of an algorithm more than the computation complexity. The communication cost for the array processors is considered under the following assumptions.

1. A node can use at most two of its incident links at a time. It can receive a message using one and send another message using the other. Furthermore, all the nodes initiate a communication at the same time, and send and receive simultaneously since the array processor is assumed to be running in SIMD fashion.

2. No time is spent to prepare a message or to wait in a queue. All packets are assumed to be of the same length and, always the same time is spent to pass any packet between neigbouring processors at any time.

These assumptions also apply well to the HAP.

An interconnection network can be represented as a graph G=(N,A) or as a topology. The nodes of a graph correspond to processors and the arcs correspond to the links. Interconnection network topologies can be assessed using the following criteria.

1. The *diameter* r of the network is the maximum distance between any pair of nodes and is measured as the number of links. For a network of diameter r, a packet can travel from one node to another in O(r) time.

2. *Arc* and *node connectivity* of a network is the number of arcs and nodes that must be deleted to make the network disconnected. The connectivity of a network is bounded above by the number of incident links to a node.

3. The *flexibility* of a network is defined as the capability of efficiently simulating algorithms designed for other topologies. To relieve the communication penalty, after dividing the main task into subtasks, each subtask must be assigned to a node considering that a node may need a value computed by another node. Such nodes must be chosen as immediate neighbours or be located as close as possible.

4. The communication *delays* (or costs) *incurred for common communication problems* like the single node and multinode broadcast problems are also important since such problems are encountered frequently in the parallel implementations of many numerical algorithms on array processors. These communication problems are explained below.

Single and multinode broadcast [4]: In the single node broadcast problem the same packet is sent from a single node to every other node. In the multinode broadcast each node broadcasts a packet to every other node.

The single node broadcast problem can be solved by constructing a spanning tree at the given node. Choosing an optimal spanning tree, one can broadcast a packet from a single node to every other node in O(r) time (,or since r is the minimum distance between the farthest nodes, a packet between the farthest nodes travels r links at most). Fig. 2.10 (a) shows by simulation how the single node broadcast problem can be solved on a two dimensional three by three mesh, and Fig. 2.10 (b) shows the corresponding spanning tree.

The multinode broadcast problem requires a spanning tree per node. However this may result in timing conflicts since some spanning trees can possess the same links. Therefore, a more complex schedule that synchronizes link possessions must be designed.

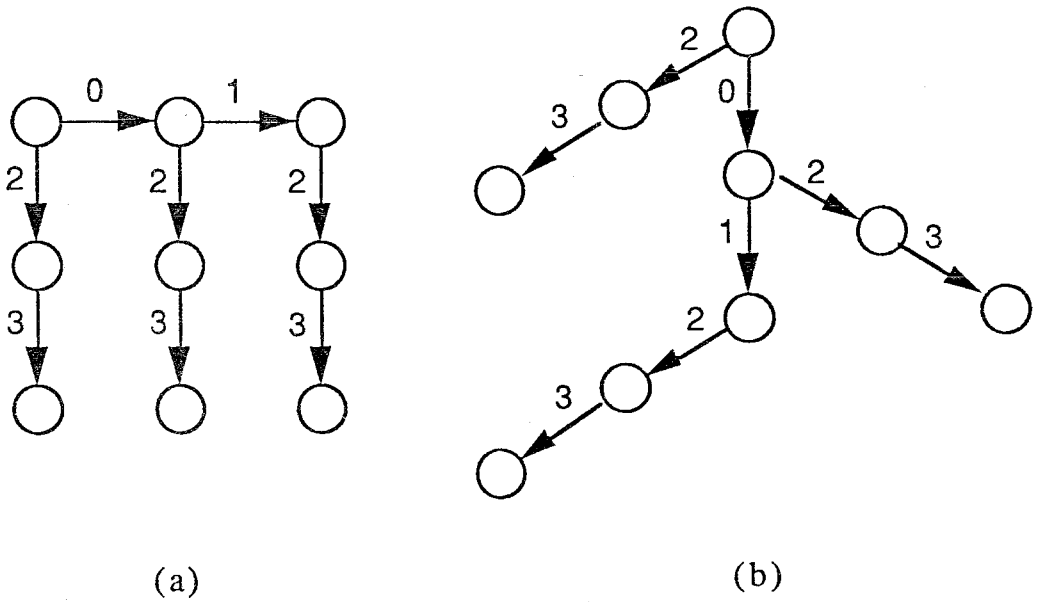


Fig. 2.10 (a) A single node broadcast on a two dimensional mesh. (b) The corresponding spanning tree.

Single and multinode accumulation [4]: In the single node accumulation, a packet from every other node is sent to a single node. These packets are combined, e.g., added, ANDed, ORed, etc., on their way to this single node. Thus, a node on the routing path may receive more than one packet but it combines these packets, and sends a single packet to the next node. A multinode accumulation is a separate single node accumulation for each node.

The single node accumulation problem can be solved by using the optimal spanning tree of the single node broadcast problem. Running the single node broadcast schedule in reverse-time, all the packets can be accumulated at a single node in O(r) time. Thus, the single node accumulation and broadcast are said to be *duals* of each other. Fig. 2.11 (a) shows the simulation of a solution to the single node accumulation problem, and Fig. 2.11 (b) shows the corresponding spanning tree.

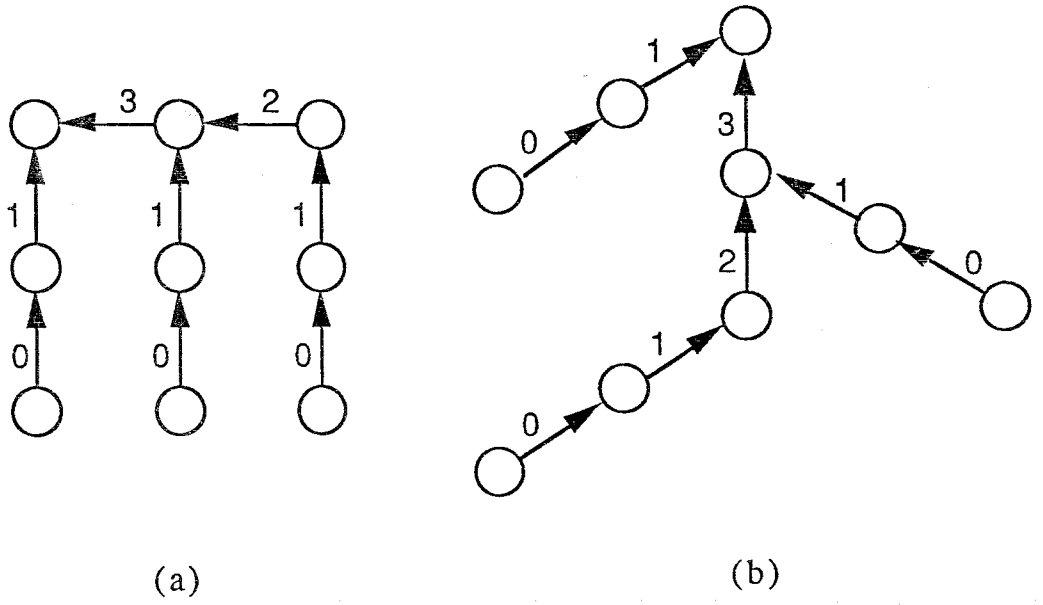


Fig. 2.11 (a) A single node accumulation on a two dimensional mesh. (b) The corresponding spanning tree.

The multinode accumulation problem can be solved by running the multinode broadcast schedule in reverse-time.

Single node scatter, single node gather, and total exchange [4]: Some algorithms require the sending of a separate packet from a single node to every other node. This is called a single node scatter problem. The dual of this problem is the separate collection of packets from every other node at a given node, and this is called a single node gather problem. Therefore, a solution to one of these is the reverse-time solution to the other. Indeed, an algorithm that schedules packet transmissions on each link and that properly takes queuing time into account is a solution to both of these problems.

In the total exchange problem, every node receives a separate packet from every other node, or, in other words, every node sends a separate packet to every other node. Therefore the the total exchange problem is the multinode version of either of the single node gather and scatter problems.

Note that in a multinode accumulation each node sends a separate packet to every other node, thereby solving the single node scatter problem. Also note that a total exchange is the generalization of a multinode broadcast if every node of the multinode broadcast sends a separate packet to every other node instead of the same packet. All of these relations are summarized in Fig. 2.12

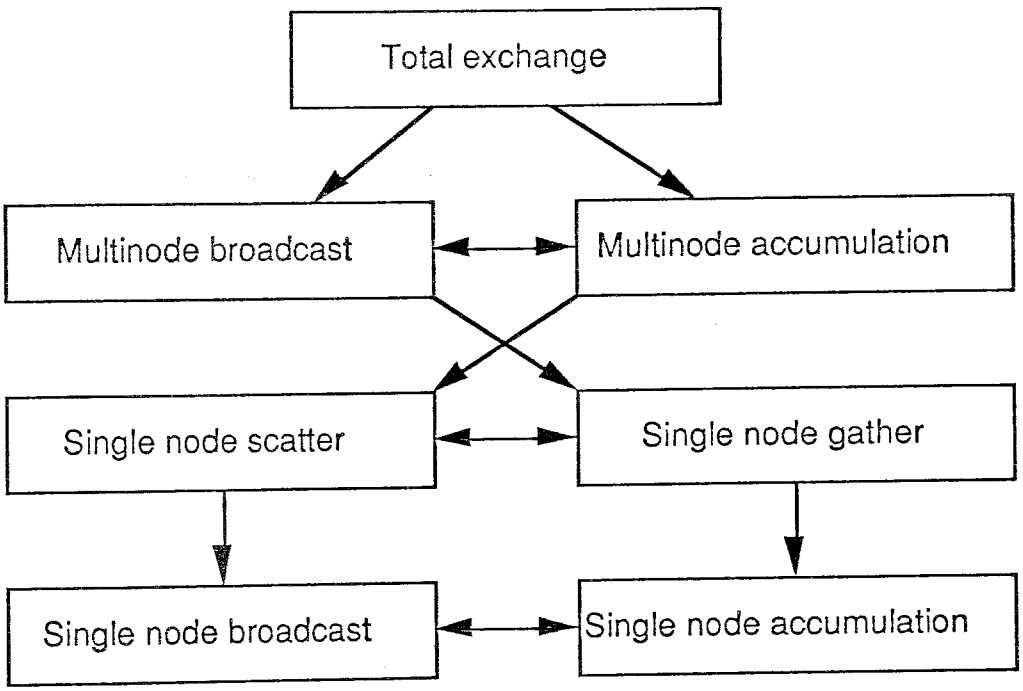


Fig. 2.12 Hierarchical ordering of basic communication problems [4].

In Fig. 2.12 a directed arc from problem x to problem y indicates that an algorithm solving x can also solve y, and the optimal time for solving y is no more than the optimal solution time of x. A bidirectional arc indicates a duality relation.

Some of the parallel algorithms described in Section 3.1 utilize a two dimensional n by n mesh array as n linear arrays of n nodes, and these algorithms include a single node accumulation and a single node broadcast.

A single node broadcast or a single node accumulation on a linear array takes only (n-1) steps or O(n) time. Fig. 2.13 illustrates a solution by simulation for the single node broadcast problem on a linear array.

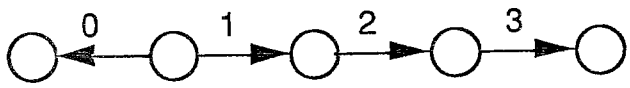


Fig. 2.13 A single node broadcast on a linear array.

2.2.5 Combined Complexities and Performance Measures

The time complexity, T_p, of an algorithm using P processors is the addition of its communication, T_{comm}, and computation, T_{comp}, complexities. An algorithm is said to be *communication bounded* if T_{comm} dominates in T_p, or *computation bounded* if T_{comp} dominates in T_p, or *unbounded* otherwise.

The *speedup* S_p of a parallel algorithm using P processors is defined as

$$T^*/T_p \tag{2.3}$$

where T^* is the optimal serial time algorithm for the same problem. S_p describes the advantage of using a parallel algorithm; it is always greater than one and less than P.

The *efficiency* E_p is defined as

$$S_p/P \tag{2.4}$$

The efficiency is a measure of processor utilization and shows the fraction of time a processor is employed. It is ideally one.

T^* can either be the time complexity of the best serial algorithm known, or the time complexity of a benchmark serial algorithm, or the simulation time of the parallel algorithm by a

single processor without any communication costs. The third definition stresses the degree an algorithm is parallelized but hides the merits of the particular algorithm in contrast with the first two.

In parallel algorithm design, there is usually a trade-off between communication and computation costs. The rule is; the less computation cost is the more is the communication cost. This is especially true for the solution of system of equations, the optimization problems and other problems of the form

$$x_i(t+1) = f(x_1(t),...,x_n(t)), \qquad i=1,..,n$$

where f is a function from $\Re^n$ to $\Re$. Solutions of this form are also known as relaxation iterations since the system relaxes, i.e., converges, after a certain number of iterations. The Hopfield model, which will be implemented in this study on a massively parallel computer, is a good example to such problems.

One can increase the concurrency of such an algorithm by dividing the subtasks into smaller subtasks. Thus more processors are made active at a time. But this leads to increased communication between processors since variables, once located in a single processor, are now shared among several processors.

The *communication ratio* is defined as T_{comm}/T_{comp}. This ratio indicates the balance between the communication and the computation complexities. For computation bounded operations like relaxation iterations high concurrency with the same communication ratio can be achieved by increasing the number of processors appropriately as the problem size increases [4].

Unfortunately, this is not true for communication bounded algorithms. As an example, consider the problem of adding N numbers using P processors on an array processor with linear array topology.

Assume that there are P processors and k.P numbers where $k \in Z^+$. First, every processor performs k additions sequentially. Then, these P results are added. The P results can be added in log(P) addition operations (See the DAG in Fig. 3.5). Therefore, the computation cost of adding N numbers is N/P+log(P) addition operations. The communication cost is (P-1), or P assuming P>>1, routing operations since this is a single node accumulation problem on a P-noded linear array. Note that all the processors add or route simultaneously, so, the total number of additions and routing operations are P times the given costs. Taking unit time for any operation, communication ratio becomes P/[N/P+log(P)]. If the number of processors are large enough, than this ratio is always greater than one, and it increases as P is increased. Decreasing P may relieve the communication penalty but lowers the speedup.

The problem of communication bound is also encountered in multiplying a vector with a matrix on a two dimensional mesh. The remedy is to use a topology other than the two dimensional mesh which will decrease the communication cost. For example, if a hypercube topology is used for a vector-matrix multiplication, then the time complexity becomes unbounded.

III IMPLEMENTATION ISSUES AND SIMULATION

In this chapter, the Hopfield artificial neural network model (see Appendix A for a description of this model) is formulated and simulated on a bit serial SIMD mesh array processor.

Section 3.1 explains and analyzes basic parallel algorithms used in the simulations, Section 3.2 analyzes complexity of the Hopfield network model on the hypothetical array processor (HAP), Section 3.3 gives the simulation results, and Section 3.4 analyzes the simulation results for speedup and efficiency.

3.1 Design of the Parallel Algorithms

Designing a parallel algorithm for an array processor is straightforward if the problem in question has the same spatial structure with the topology of the array processor at hand. Also, it is better if the interactions between the variables and equations of the problem are spatially local. The outer product of two vectors, which is used in the Hopfield model, satisfies both of these conditions. Parallel implementations of such solution techniques yield the highest, if not ideal, speedups and efficiencies.

The violation of the first condition above requires the design of an algorithm that maps the spatial structure of the algorithm to the topology of the array processor. The second condition can be violated as a result of the violation of the first condition. If the second condition is violated, then special algorithms that accomplish the required interprocessor communications like the spanning trees of Section 2.2.4 must be developed. The violation of the second condition usually causes a parallel algorithm to be communication

bounded. The algorithms, which include matrix operations, like the class matching part of the Hopfield network, usually violate the second condition.

The parallel algorithms for vector and matrix storage, outer product of two vectors, transposition of a block, and multiplication of a vector with a square matrix have been used in the Hopfield network simulation.

3.1.1 Vector Storage

Before describing how vectors are stored on the Blitzen array processor in the simulations of this study, we first define a structure which will be referred to as a *block*.

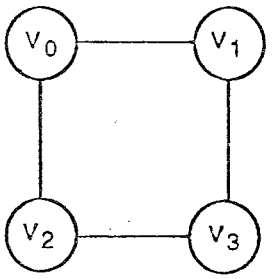
A block is a set of P pieces of local (or VRAM but not both) memory beginning at the same address. Moreover, every element of a block consists of the same number of bits and, each belongs to a different processor. Thus, a block is determined by its type, i.e. local or VRAM block, an address and a number of bits. A number is said to be at block *B* if it is stored at the address of block *B* using the same number of bits. The term *block of numbers* refers to P numbers stored at a block. If one needs to store more than P numbers, e.g. a long vector or a large matrix, then he/she may use multiple blocks (not necessarily but preferably) at consecutive addresses. In the expressions given below, $\lfloor x/y \rfloor$ is the greatest integer less than or equal to x/y, and x%y is the remainder of x/y.

A vector of N elements is stored using

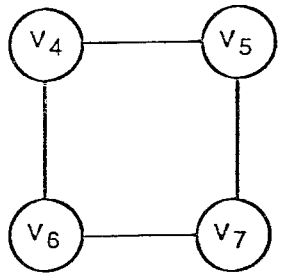
$$\lfloor (N-1)/P \rfloor + 1 \qquad (3.1)$$

blocks. A vector element v_i, for i=0 to K-1, is stored in processing element (PE) $(\lfloor (i\%P)/\sqrt{P} \rfloor, (i\%P)\%\sqrt{P})$ at the block number

$$\lfloor i/P \rfloor \qquad (3.2)$$



(a) Block number 0.

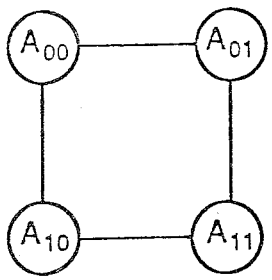


(b) Block number 1.

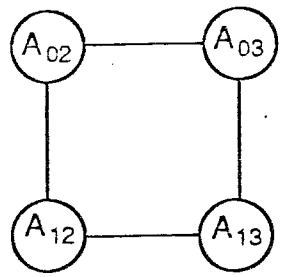
Fig. 3.1 The storage of an 8-tupled vector on a 2 by 2 array.

For example, a vector of 8 elements can be stored using 2 blocks on a 2 by 2 array. Fig. 3.1 depicts this case.

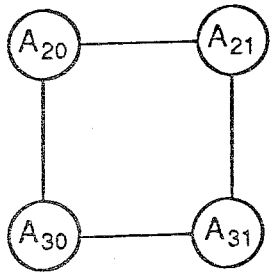
3.1.2 Matrix Storage



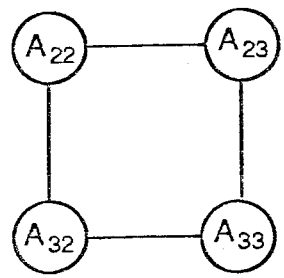
(a) Block number 0.



(b) Block number 1.



(c) Block number 2.



(d) Block number 3.

Fig. 3.2 The storage of a 4 by 4 matrix on a 2 by 2 array.

An M by N matrix is stored using

$$(\lfloor (M-1)/\sqrt{P} \rfloor + 1)(\lfloor (N-1)/\sqrt{P} \rfloor + 1) \qquad (3.3)$$

blocks. A matrix element A(i,j), for i=0 to M-1, j=0 to N-1, is stored in PE (i%P,j%P) at the block number

$$\lfloor i/\sqrt{P} \rfloor (\lfloor (N-1)/\sqrt{P} \rfloor + 1) + \lfloor j/\sqrt{P} \rfloor \qquad (3.4)$$

For example, a 4 by 4 matrix can be stored using 4 blocks on a 2 by 2 array. Fig. 3.2 shows this case.

3.1.3 Transposition of a Matrix

The interconnection network of Blitzen is very suitable for the transposition of a matrix since it provides routing in the diagonal directions, and it also allows routing through the PEs on the edges by connecting North-South and East-West edges. Such an interconnection network is called a *wrap-around X-grid,* and routing through such a topology is called *wrap-around routing*. If the other mode of routing, called *grid routing,* is enabled, then the edges are disconnected. The processors on the incoming edge receive zeros in grid routing.

The algorithm given in Fig. 3.3 transposes a block of numbers utilizing 6 blocks. One of the blocks, called the *original block,* keeps the block of numbers to be transposed, and another block, called the *transposed block,* keeps the transposed form of the original block.

The numbers at the original block are divided into two groups as the northeast (NE) and southwest (SW) travellers. An NE (SW) *traveller* is a number whose transposed position is in the northeast (southwest) direction relative to its original position. Each group has *a pair* of blocks. The numbers at the original block are copied to the first block, called the travellers block *(travellerNE* and *travellerSW)*, of each pair at the beginning. The second block of numbers, called the passport block *(passportNE* and *passportSW)*,

are initialized by subtracting the row index of a PE from its column index for the NE travellers and by subtracting the column index of a PE from its row index for the SW travellers.

The transposition operation is accomplished in $\sqrt{P}$ steps for each traveller group. At the beginning of each step, the numbers associated with a zero passport value is copied to the transposed block. After incrementing (decrementing) the numbers at the passport block for the NE (SW) travellers, the NE (SW) travellers and their passports are routed one PE northeast (southwest), thereby assigning new PEs to each of the travellers.

If an instruction is indented to the right after a "FOR EVERY PE (i,j)" sign, then this instruction is executed by all the PEs in parallel. Such instructions end with a semi-colon. The variables in the algorithm given in Fig.3.3 refer to the numbers stored at the above mentioned blocks in the PE (i,j) they belong to (e.g. travellerNE is the number stored at the travellerNE block in PE (i,j) when it takes place in an instruction). Instructions containing variables are always indented to the right. Note that the letters *i* and *j* are exclusively used for PE indexing in all of the algorithms and they change from 0 to $\sqrt{P}$-1. The parallel algorithm for the transposition of a $\sqrt{P}$ by $\sqrt{P}$ matrix is given in Fig. 3.3 in the pseudo-language of the HAP.

The notation used to express the time complexity, speedup, and efficiency of an algorithm is given below.

$T_p(x,N)$: The time complexity of a parallel algorithm x using P processors for a problem size N.

$T_s(x,N)$: The time complexity of a sequential algorithm x for a problem size N.

$S_P(x,N)$: As defined in Section 2.2.5, speedup of an algorithm x for a problem size N.

$E_P(x,N)$: As defined in Section 2.2.5, efficiency of an algorithm of x using P processors for a problem size N.

f(x)=O(g(x)): As defined in Section 2.2.3. We will use this notation due its convenience for us. That is, by showing that a problem can be solved with a property expressed as g(x), we will immediately be obtaining an upper bound in the order of g(x) for this property.

```
SetRoute(torus);
FOR EVERY PE (i,j)
    travellerNE = original;
    passportNE = j-i;
    travellerSW = original;
    passportSW = i-j;
    For (k=0 to √P -1){
        If (passportNE = 0) then
          transposed = travellerNE;
        passportNE = passportNE +1;
        SendToNorthEastReceiveFromSouthWest
          (travellerNE,travellerNE);
        SendToNorthEastReceiveFromSouthWest
          (passportNE,passportNE);
        If (passportSW = 0) then
          transposed = travellerSW;
        passportSW = passportSW -1;
        SendToSouthWestReceiveFromNorthEast
          (travellerSW,travellerSW);
        SendToSouthWestReceiveFromNorthEast
          (passportSW,passportSW);
    }
```

Fig. 3.3 A parallel algorithm.for transposition of a $\sqrt{P}$ by $\sqrt{P}$ matrix.

The problem size for the transposition problem can be defined as the number of elements of the matrix in question. Therefore, this parallel transposition algorithm yields

$$T_p \text{ (transposition,P)} = O(\sqrt{P}) \qquad (3.5)$$

compared to the

$$T_s \text{ (transposition,P)} = O(P) \qquad (3.6)$$

of a sequential transposition algorithm.

3.1.4 Outer Product of a Vector with Itself

The outer product of a vector of size M and a vector of size N results in an M by N matrix. Such a matrix is constructed in the weight matrix generation part of the Hopfield network. Although the storage of such a matrix may cause excessive memory requirements, the huge size of the images makes the simulation of the Hopfield network attractive on a SIMD array processor.

The algorithm given in Fig. 3.4 describes the outer product of a vector of P elements with itself (however, it can easily be generalized to the case of any size). Since every element must be multiplied with every other element, the parallel version of the outer product algorithm includes a multinode broadcast. After the multinode broadcast, every PE will have the whole input vector, and PE (i,j), for i,j=0 to $\sqrt{P}$-1, will generate the row iP+j of the output matrix by performing P multiplications.

Unfortunately, it may be impossible to implement the outer product as described above since a PE may not have enough local memory to store an input vector or a single row of the output matrix. Therefore, this procedure needs to be revised to account for the limited local memory of a PE. A revised procedure may receive

the input vector elements one by one and may swap the result to the VRAM after each multiplication.

The algorithm given below assumes that a PE has enough memory to hold the $\sqrt{P}$ elements of the original vector. It solves the multinode broadcast problem by performing $\sqrt{P}$ single node broadcasts in parallel on $\sqrt{P}$ sized linear arrays $2\sqrt{P}$ times. Thus, each PE receives the other operand of every multiplication just before that multiplication. Each PE, performing P multiplications to find the outer product of two $\sqrt{P}$-tupled subvectors, generates P elements of the output matrix.

The algorithm utilizes $4+\sqrt{P}+P$ blocks. One of the blocks, called the *original* block, keeps the input vector in its original form, and another block, called the *transposed* block, keeps the original block of numbers transposed as a $\sqrt{P}$ by $\sqrt{P}$ matrix. The next $\sqrt{P}$ blocks is a series of blocks called the *broador* blocks. The broador block i, for i=0 to $\sqrt{P}$-1 stores the elements from $i\sqrt{P}$ to $(i+1)\sqrt{P}$ of the input vector. A third kind of block, called the *broadlast* block, stores the last number broadcast from the transposed block, and, finally, the *resultant* block keeps the result of a multiplication (a broador block multiplied by the broadlast block). After each multiplication, the resultant block is saved to a certain VRAM block. These P number of VRAM blocks are called the *matrix* blocks.

The input vector and the resultant matrix are stored as explained in Sections 3.1.1 and 3.1.2. Sometimes, in the algorithms, only an operation and the input/output blocks are given instead of the instructions just for the simplicity. Such operations are called *block operations*. They also activate all or, if mentioned, only designated PEs. The HAP accomplishes executing just a group of PEs by controlling the mask register in every PE. The outer product algorithm given in Fig. 3.4 is written in the pseudo-language described in Section 2.2.1.

The problem size for an outer product can be defined as the input vector size. A sequential outer product algorithm yields

$$T_s \text{ (outer product,P)} = O(P^2) \qquad (3.7)$$

as a direct reflection of its computation cost which is P^2 multiplications.

```
Transpose the numbers in the original block to the
  transposed block.
SetRoute(grid);
For (k=0 to √P-1){
    Let the PEs at the k^th row broadcast the numbers at
      the original block to the broador block k of the
      PEs in the same column.
}
For (k=0 to √P-1){
    Let the PEs at the k^th column broadcast the
      numbers at the transposed block to the broadlast
      block of the PEs at the same row.
    For (q=0 to √P-1){
        FOR EVERY PE (i,j)
            result=broador[q]*broadlast;
            matrix[k*√P+q] = result;

    }
}
```

Fig. 3.4 A parallel algorithm for the outer product of a vector with itself.

The computation cost of the algorithm given in Fig. 3.4 is just P multiplications. Its communication cost is 2P routing operations computed as $2\sqrt{P}$ single node broadcasts on $\sqrt{P}$-noded linear arrays. The transposition operation for the problem size of $\sqrt{P}$ contributes a complexity of $O(\sqrt{P})$. Therefore, this algorithm is unbounded with a time complexity given as,

$$T_P \text{ (outer product,P)} = O(P) + O(2P)+O(\sqrt{P}) \qquad (3.8)$$
$$= O(P)$$

Note that we can easily extend the analysis and algorithm given above to the case where the outer product of a $k\sqrt{P}$-tupled vector ($k \in Z^+$ and $k \leq \sqrt{P}$) is obtained using P processors.

In this case, each PE performs k^2 multiplications to find the outer product of two k-tupled vectors, and thus, generates k^2 elements of the $k\sqrt{P}$ by $k\sqrt{P}$ output matrix. For the communication between the PEs, k single node broadcasts are performed in parallel on $\sqrt{P}$ sized linear arrays 2k times. The cost due the transposition operation remains the same since we again need to transpose a block of numbers. Thus, we obtain the following time complexity for the parallel implementation of the transposition problem,

$$T_P \text{ (outer product,}k\sqrt{P}) = O(k^2) + O(2k\sqrt{P}) + O(\sqrt{P}) \qquad (3.9a)$$

(3.9a) can be rewritten as follows since $k\sqrt{P} \geq \sqrt{P}$ and "2" in $(2k\sqrt{P})$ of (3.9a) is a constant.

$$T_P \text{ (outer product,}k\sqrt{P}) = O(k^2 + k\sqrt{P}) \qquad (3.9b)$$

The time complexity of the sequential implementation of this case is given as

$$T_s \text{ (outer product,}k\sqrt{P}) = O(k^2P) \qquad (3.10)$$

Note that the outer product algorithm given above, is unbounded or communication bounded depending on the value of k.

3.1.5 Multiplication of a Matrix with a Vector

First, we will show that the HAP can multiply a $\sqrt{P}$-tupled vector v with a $\sqrt{P}$ by $\sqrt{P}$ matrix *A* with a time complexity of $O(\sqrt{P})$. Then, assuming that we have a P-tupled vector and P by P matrix,

we will show these two can be multiplied in $O(P\sqrt{P})$. Finally, we will show that a $k\sqrt{P}$-tupled vector can be multiplied with a $k\sqrt{P}$ by $k\sqrt{P}$ matrix in $O(k^2\sqrt{P})$ time where $k \in Z^+$ and $k \leq \sqrt{P}$.

The multiplication of a vector of $\sqrt{P}$ elements with a $\sqrt{P}$ by $\sqrt{P}$ matrix consists of $\sqrt{P}$ inner products. This can be accomplished the HAP by dividing its P nodes (PEs) into rowwise $\sqrt{P}$ noded $\sqrt{P}$ linear arrays.

The inner product of two $\sqrt{P}$-tupled vectors consists of $\sqrt{P}$ multiplication operations and the addition operations needed to take the sum of the $\sqrt{P}$ multiplication results. Fig. 3.5 gives a DAG G that can be used to accomplish these $\sqrt{P}$ additions.

The depth of G of Fig. 3.5 is $\log(\sqrt{P})$. A schedule S using $\sqrt{P}$ processors can finish this task in $\log(\sqrt{P})$ steps; it uses $1/2*\sqrt{P}$ processors at the first step, $1/2^n*\sqrt{P}$ processors at the n^{th} step, and a single processor at the $\log(\sqrt{P})^{th}$ or final step. This G and S pair is optimal because of the property 1 of Section 2.2.3 which states that

$$T_\infty \geq \log(\sqrt{P}) \qquad (3.11)$$

where T_∞ is the time complexity that can be achieved using infinitely many processors. Therefore, the computation complexity of the parallel addition algorithm, represented by the above G and S pair, is minimum and is equal to $O(\log(\sqrt{P}))$.

Since a linear array topology is used, we need an interprocessor communication scheme to enable message (result) communication between the PEs of the linear array. A solution to this communication problem is given by the algorithm described in Section 2.2.4, and this is the solution to the single node accumulation problem on a linear array.

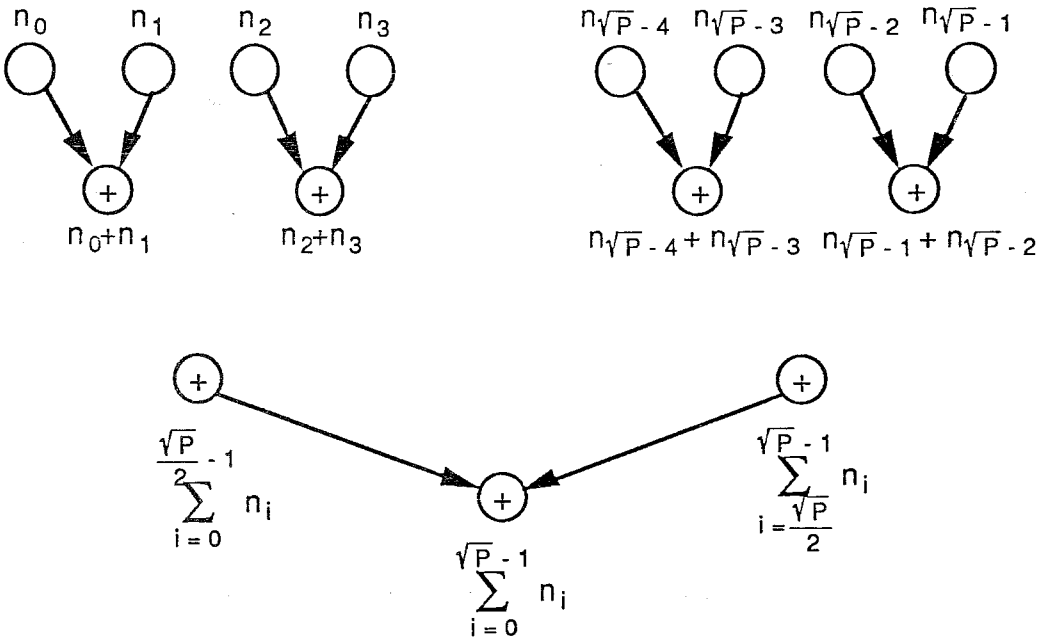


Fig. 3.5 A DAG to add $\sqrt{P}$ numbers.

It was shown in Section 2.2.4 that the single node accumulation problem requires at most $(\sqrt{P}-1)$ routing operations on a linear array of $\sqrt{P}$ nodes. Therefore, the communication complexity of realizing the above parallel addition algorithm on a linear array of $\sqrt{P}$ nodes is $O(\sqrt{P})$. Then, the total complexity of the parallel addition algorithm is $O(\sqrt{P})+O(\log(\sqrt{P}))$, and this leads to a time complexity of $O(\sqrt{P})$ for the addition of $O(\sqrt{P})$ numbers on a $\sqrt{P}$-noded linear array.

Assigning an element of each vector to each of the PEs, all of the multiplications can be completed in a *single step*. Therefore, the time complexity of performing an inner product of two $\sqrt{P}$-tupled vectors on a $\sqrt{P}$-noded linear array is $O(\sqrt{P})$.

Let's assume that at the beginning of the matrix-vector multiplication, j^{th} PE on linear array i, for i,j=0 to $\sqrt{P}-1$ stores two vector elements (the element (i,j) of the matrix and the element j of the vector). Then, the linear array i computes the element i of the resultant vector with a time complexity of $O(\sqrt{P})$. Since all the linear arrays run in parallel, the resultant vector can be computed

with a time complexity of $O(\sqrt{P})$. Therefore, a $\sqrt{P}$-tupled vector v can be multiplied with a $\sqrt{P}$ by $\sqrt{P}$ matrix A in a time complexity of $O(\sqrt{P})$ using a mesh array processor of P number of PEs. For a square matrix-vector multiplication (SMVM) the problem size can be defined as the dimension of the input vector. Therefore,

$$T_P\,(SMVM,\sqrt{P}) = O(\sqrt{P}) \qquad (3.12)$$

The next task is the multiplication of a P-tupled vector (the input image to the Hopfield network) with a P by P matrix (the weight matrix) on a mesh array processor of P processors. Therefore, this task requires P number of vector inner products with a vector of size P or $P\sqrt{P}$ subvector inner products (neglecting addition operations needed to take the sum of the subvector inner products) for a vector of size $\sqrt{P}$. Since the HAP can perform $\sqrt{P}$ inner products of size $\sqrt{P}$ in parallel with a time complexity of $O(\sqrt{P})$, the total time complexity is given as

$$T_P\,(SMVM,P) = [(P\sqrt{P})/\sqrt{P}]O(\sqrt{P}) = O(P\sqrt{P}) \qquad (3.13)$$

Now, we will define the terminology for the blocks used in the matrix vector multiplication algorithm given in Fig. 3.6. All blocks are local RAM blocks unless otherwise stated. The P-tupled input vector and the P by P input matrix are stored on the HAP as described in Sections 3.1.1 and 3.1.2. Therefore, a single block, called the *vector* block, must be used to store the input vector (image) v and, P number of VRAM blocks, called the *matrix* blocks, must be used to store the input (weight) matrix A.

The *tempmat* block stores the matrix block currently swapped from VRAM. The *transposed* block stores the numbers at the vector block as transposed i.e. the numbers at the vector block are considered as the elements of a $\sqrt{P}$ by $\sqrt{P}$ matrix, and then transposed. The *broadcast* block stores the numbers currently broadcast (a single node broadcast on a $\sqrt{P}$ noded linear array) from the *transposed* block. The *multiplied* block stores one to one multiplication of tempmat and broadcast blocks. The *addrow* block stores the addition (a single node accumulation on a $\sqrt{P}$-noded

linear array) of all the numbers at a row of the multiplied block. Finally, the *resultant* block stores the P-tupled output vector.

The parallel algorithm for SMVM given in Fig. 3.6 is written in the pseudo-language of the HAP. The instructions indented to the right after an "ONLY FOR PEs (i,j)" sign are only executed at the designated PEs. Such instructions also end with a semi-colon. If an "ONLY FOR PEs (i,j)" sign comes after a "FOR EVERY PE (i,j)" sign, the former dominates the with reference to "ONLY ..." indented instructions.

In the algorithm given in Fig. 3.6, a transposition operation with a problem size of P is performed once, a single node broadcast operation is performed $\sqrt{P}$ times, a multiplication operation is performed P times, a single node accumulation is performed P times, and an addition operation is performed P times. Therefore, the time complexity, T_P, of the parallel algorithm of Fig. 3.6 is given as,

$$T_P(\text{SMVM algorithm},P) = O(\sqrt{P}) + \sqrt{P}O(\sqrt{P}) + PO(1) + PO(\sqrt{P}) + PO(1) \qquad (3.14)$$

hence,

$$T_P(\text{SMVM algorithm},P) = O(P\sqrt{P}) \qquad (3.15)$$

Note that this result is the same as the expression (3.13). A sequential algorithm accomplishing the same task gives

$$T_s(\text{SMVM},P) = O(P^2) \qquad (3.16)$$

Note that the SMVM algorithm given above is communication bounded since the time complexity of the single node accumulation accomplished during an inner product governs its time complexity.

```
Transpose the input vector into the transposed block.
For (k=0 to √P-1){
    Let the PEs at the k^th column broadcast the
      numbers at the transposed block to the PEs at
      the same row.
    For (q=0 to √P-1){
        FOR EVERY PE (i,j)
            tempmat = matrix[q.(√P - 1) + k];
            multiplied = tempmat * broadcast;
        Add all the numbers at a row of the multiplied
          block into the addrow block.
        ONLY FOR PEs (i,q)
            resultant= resultant + addrow;
    }
}
```

Fig. 3.6 A parallel algorithm for (square) matrix-vector multiplication.

Now, it is easier to show the more general case of $k\sqrt{P}$ by $k\sqrt{P}$ matrix and $k\sqrt{P}$-tupled vector multiplication where $k \in Z^+$ and $k \leq \sqrt{P}$. In this case, we need to perform $k\sqrt{P}$ number of vector inner products for a vector size of $k\sqrt{P}$, or $k^2\sqrt{P}$ number of subvector inner products (neglecting addition operations needed to take the sum of the subvector inner products) for a vector size of $\sqrt{P}$. Since the HAP can perform $\sqrt{P}$ inner products of size $\sqrt{P}$ with a time complexity of $O(\sqrt{P})$, the time complexity of this case is given as,

$$T_P(SMVM, k\sqrt{P}) = [(k^2\sqrt{P})/\sqrt{P}]O(\sqrt{P}) = O(k^2\sqrt{P}) \qquad (3.17)$$

The time complexity for the sequential implementation of this case is as follows,

$$T_s(SMVM, k\sqrt{P}) = O(k^2P) \qquad (3.18)$$

3.2 Complexity of the Hopfield Network

The Hopfield network algorithm consists of two parts. In the first part, called the *weight matrix generation* (WMG) part, a matrix *(weight matrix)* is generated from a set of input vectors called the *image classes*. In the second part, called the *class matching* (CM) part, another input vector, called the *input image*, is tried to be matched with one of the image classes. Note that the input image and an image class has the same number of pixels.

For the following three sections, assume that the input image has $k\sqrt{P}$ pixels ($k \in Z^+$, $k \leq \sqrt{P}$, $1 < \sqrt{P}$). The problem size (N) is defined as the number of pixels of an image.

3.2.1 Weight Matrix Generation

The WMG part consists of the outer product of each class image class vector with itself, the addition of the resultant matrices and the resetting, i.e. making zero, of the diagonal elements of the final matrix. Assuming that there are C image classes ($1 \leq C < \sqrt{P}$) and using (3.10), the time complexity of the WMG's sequential implementation is

$$T_s(WMG,k\sqrt{P}) = CO(k^2P) + (C-1)O(k^2P) + O(k\sqrt{P}) \qquad (3.19a)$$

where the first term in the above expression refers to the outer product, the second term to the addition of the outer products, and the third term to the resetting of the diagonal elements.Since $k^2P \geq P$ and $C>1$, the following can be obtained using (3.19a),

$$T_s(WMG,P) = CO(k^2P) \qquad (3.19b)$$

Based on the arguments given in Section 3.1.4, the time complexity of the outer product of $k.\sqrt{P}$-tupled vectors on the hypothetical array processor (HAP) is $O(k^2+k\sqrt{P})$. The addition of the outer products have a time complexity of (C-1)O(1) since the

addition of these matrices are accomplished by all the PEs in parallel. Resetting of the diagonal elements of the weight matrix has a time complexity of O(k) since $\sqrt{P}$ PEs can perform the resetting operation simultaneously. Therefore, a weight matrix can be generated with a time complexity of,

$$T_P(WMG,k\sqrt{P}) = CO(k^2+k\sqrt{P}) + (C-1)O(1) + O(k) \qquad (3.20a)$$

Since $(k^2+k\sqrt{P})>k$ and C>1, (3.20a) can be rearranged as follows,

$$T_P(WMG,k\sqrt{P}) = CO(k^2+k\sqrt{P}) \qquad (3.20b)$$

Up until now, we have expressed the problem size (N) in terms of the number of processors, P. Now, we will derive the time complexity, speedup, and efficiency expressions for the WMG without expressing N in terms of P.

Since we have previously assumed that $N=k\sqrt{P}$, we can substitute $k=N/\sqrt{P}$ into expressions (3.19b) and (3.20b). Thus, the following expressions can be obtained,

$$T_s(WMG,N) = CO(N^2) \qquad (3.21)$$

$$T_P(WMG,N) = CO(N^2/P + N) \qquad (3.22)$$

The speedup and efficiency for the parallel implementation of the WMG can be obtained as follows.

$$S_P(WMG,N) = O(N/(N/P + 1)) \qquad (3.23)$$

$$E_P(WMG,N) = O(N/(N + P)) \qquad (3.24)$$

For the case N=P, in which a single pixel is assigned to each processor, the following expressions are obtained,

$$T_s(WMG,P) = CO(P^2) \qquad (3.25)$$

$$T_P(WMG,P) = CO(P) \quad (3.26)$$

$$S_P(WMG,P) = O(P) \quad (3.27)$$

$$E_P(WMG,P) = O(1) \quad (3.28)$$

3.2.2 Class Matching

The second part of the Hopfield algorithm is a series of iterations, called *class matching iterations,* repeated until convergence. In each class matching iteration, the input image vector is multiplied with the weight matrix. The *resultant vector* passes through a nonlinear function to generate the *output image.* If the output image vector is not the same as the input image vector, then it is taken as the new input image vector; otherwise, the iterations are stopped.

Since a class image has $k\sqrt{P}$ pixels, the weight matrix is a $k\sqrt{P}$ by $k\sqrt{P}$ matrix. As discussed in Section 3.2.5, the time complexity of the multiplication of a $k\sqrt{P}$ by $k\sqrt{P}$ matrix with a $k\sqrt{P}$-tupled vector on a sequential processor is $O(k^2P)$. The application of a nonlinear function to the $k\sqrt{P}$-tupled resultant vector requires $k\sqrt{P}$ applications. The comparison of the $k\sqrt{P}$-tupled input and output image vectors also requires $k\sqrt{P}$ comparisons. Therefore, the application of a nonlinear function and the comparison of the $k\sqrt{P}$-tupled input and the output image vectors present the same time complexity, which is $O(k\sqrt{P})$, for a sequential implementation. Adding the time complexities of the matrix-vector multiplication, application of the nonlinear function, and the comparison of the input and output image vectors, the time complexity of the sequential implementation of a class matching iteration is found as follows,

$$T_s(CM,k\sqrt{P}) = O(k^2P) + O(k^2P) + O(k\sqrt{P}) = O(k^2P) \quad (3.29)$$

since $k^2P > k\sqrt{P}$.

In Section 3.1.4, we have shown that the time complexity of the multiplication of a $k\sqrt{P}$ by $k\sqrt{P}$ matrix with a $k\sqrt{P}$-tupled vector on the HAP is $O(k^2\sqrt{P})$. The application of a nonlinear function to the $k\sqrt{P}$-tupled resultant vector and the comparison of the $k\sqrt{P}$-tupled input and the output image vectors operations have O(1) time complexities since $k\sqrt{P}$ PEs perform these in parallel. Therefore, the time complexity of a single class matching iteration is given as,

$$T_P\,(CM,k\sqrt{P}) = O(k^2\sqrt{P}) + O(1) + O(1) = O(k^2\sqrt{P}) \qquad (3.30)$$

Now, it is trivial to derive the expressions for the time complexities, speedup, and efficiency of a single class matching iteration in terms of N and P. Substituting $k=N/\sqrt{P}$ into (3.29) and (3.30), the following expressions can be obtained,

$$T_s\,(CM,N) = O(N^2) \qquad (3.31)$$

$$T_P\,(CM,N) = O(N^2/\sqrt{P}) \qquad (3.32)$$

The speedup and efficiency for the parallel implementation of the CM is found as follows,

$$S_P\,(CM,N) = O(\sqrt{P}) \qquad (3.33)$$

$$E_P\,(CM,N) = O(1/\sqrt{P}) \qquad (3.34)$$

For the case N=P, in which a single pixel is assigned to each processor, the following expressions are obtained,

$$T_s\,(CM,P) = O(P^2) \qquad (3.35)$$

$$T_P(CM,P) = O(P\sqrt{P}) \qquad (3.36)$$

$$S_P(CM,P) = O(\sqrt{P}) \qquad (3.37)$$

$$E_P(CM,P) = O(1/\sqrt{P}) \qquad (3.38)$$

Note that the implementation of the class matching on the HAP is communication bounded since the matrix-vector multiplication governs T_P.

3.2.3 The Complete Hopfield Algorithm

Assuming that the Hopfield algorithm (HOP) converges in R iterations, using (3.21) and (3.31) a sequential implementation of the Hopfield algorithm gives a time complexity of,

$$T_s(HOP,N) = CO(N^2) + RO(N^2P) \tag{3.39}$$

Based on the complexities given in (3.22) and (3.32), our implementation of the Hopfield algorithm on the HAP has a time complexity

$$T_p(HOP,N) = CO(N^2/P+N) + RO(N^2/\sqrt{P}) \tag{3.40}$$

Now, assume that we are given a set of image classes and a certain input image. Thus, C and R can be taken as constants. Under this assumption, the time complexities of the sequential and parallel implementations are given as

$$T_s(HOP,N) = O(N^2) \tag{3.41}$$

$$T_p(HOP,N) = O(N^2/P+ N) + O(N^2/\sqrt{P}) \tag{3.42a}$$

(3.42a) can be written as follows since $N^2/\sqrt{P} > N^2/P$,

$$T_p(HOP,N) = O(N^2/\sqrt{P} + N) \tag{3.42b}$$

The speedup and efficiency for the overall implementation of the Hopfield network model can be obtained from (3.41) and (3.42b) as follows,

$$S_p(HOP,N) = O(1/(1/\sqrt{P} +1/N)) \tag{3.43}$$

$$E_P\,(HOP,N) = O(1/(\sqrt{P} + P/N)) \qquad (3.44)$$

For the case N=P, in which a single pixel is assigned to each processor, the following expressions are obtained,

$$T_s\,(HOP,N) = O(N^2) \qquad (3.45)$$

$$T_p\,(HOP,P) = O(P\sqrt{P}) \qquad (3.46)$$

$$S_P\,(HOP,P) = O(\sqrt{P}) \qquad (3.47)$$

$$E_P\,(HOP,P) = O(1/\sqrt{P}) \qquad (3.48)$$

From the above expressions, one can deduce that for a problem size large enough, the greater the number of processors, the higher is the speedup. However, implementation of the Hopfield algorithm on the HAP is communication bounded and the processor utilization decreases as the number of processors increases. Also note that, the parallel algorithm for the Hopfield network model has similar time complexity, speedup, and efficiency expressions with a single class matching iteration for large problem sizes. Therefore, the performance of the parallel implementation of the Hopfield network model is governed by the class matching iterations.

3.3 Simulation Results

The parallel Hopfield network algorithm, which is formulated in Section 3.2.3, is implemented on a software simulator, called the Blitzen simulator. This software package simulates a bit serial SIMD mesh array processor, the Blitzen massively parallel processor (BMPP), which is actually very similar to the HAP (see Appendix B for the BMPP the Blitzen simulator).

The algorithms described in Section 3.1 are simulated on the Blitzen simulator. The simulations provided data about the actual running times of these algorithms since the simulator returns the number of clock (processor) cycles of the BMPP at the end of a simulation. One can easily convert the number of processor cycles into seconds, or any units of time, if the clock cycle of a PE is known. For example, if a simulation takes $10*10^5$ cycles, and if the processor speed is 20 Mhz, then the real running time on the BMPP will be

$$10*10^5/20*10^6 = 0.05 \text{ seconds}$$

The Blitzen simulator allows the utilization of any processor array size ranging from 32 by 32 to P by P where P is a power of 2. Therefore, the code, which simulates the Hopfield network model, (see Appendix C for the code) is designed for the utilization any processor array size. However, due to the limitations on the computer (DEC's DS5500 with a 32 MB RAM), where the Blitzen simulator was installed, only 32 by 32 (1K PEs) and 64 by 64 (4K PEs) array sizes were simulated for this study.

It is thought that only two array sizes would not be enough to show the effect of array size on the number of cycles in a simulation study. Therefore, we searched for a method to obtain the number of cycles without actually running the Blitzen simulator. We have succeeded to derive three functions, called the *cycle estimation functions for the BMPP* (see Appendix D for these

functions and their derivations), which return the number of processor cycles it takes to simulate a Hopfield network model and its weight matrix generation (WMG) and class matching (CM) parts on the Blitzen simulator.

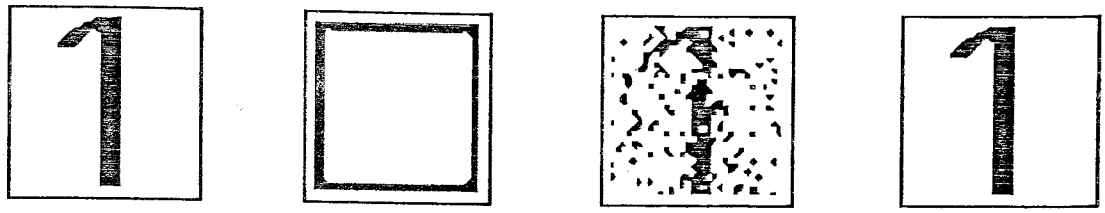
The cycle estimation functions for the BMPP were derived after a careful examination of the code written for the actual simulations. These functions have the problem size, number of classes, number of iterations, number of PEs, and number of bits used to represent a pixel, a weight matrix element, and a PE index as their arguments. These functions return the same number of cycles as the Blitzen simulator for the actual simulations of the cases for 1K and 4K PEs.

The results obtained with the BMPP have to be compared with the results of a sequential machine in order to evaluate the performance of this parallel architecture. Therefore, we have designed the *imaginary sequential processor* (ISP). Then, we have derived *cycle estimation functions for the ISP* similar to the BMPP cycle estimation functions (see Appendix E for the ISP, cycle estimation functions for the ISP and their derivations). The ISP cycle estimation functions return the number of processor cycles it takes to simulate a Hopfield network model and its WMG and CM parts on the ISP.

Section 3.3.1 gives the simulation results obtained by running the Blitzen simulator and using the cycle estimation functions for the BMPP, and Section 3.3.2 gives the number of cycles it takes to simulate the Hopfield network models with different problem sizes on the ISP.

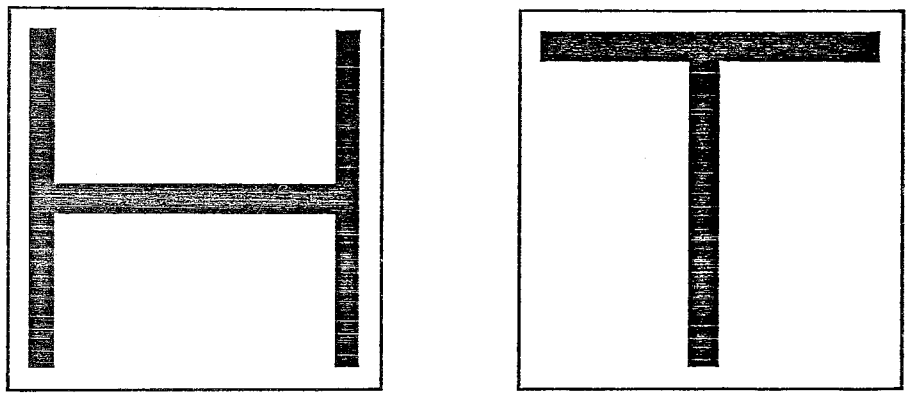
3.3.1 Results for the BMPP

Simulation Results on the Blitzen Simulator: Each simulation had two image classes. All the input images were distorted ten percent with a uniformly distributed probability density function. All simulations converged in 2 iterations. The problem size is defined as the number of pixels of an image.

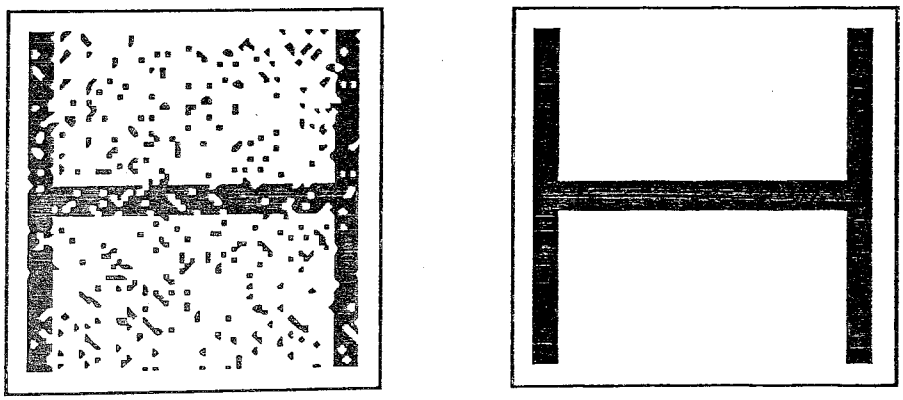


(a) (b) (c) (d)

Fig. 3.7 Images of the simulation test with 1K PEs (each image has 1024 pixels). (a) Image class number 1. (b) Image class number 2. (c) Distorted input image.(d) Output image obtained after convergence.



(a) (b)



(c) (d)

Fig. 3.8 Images of the simulation test with 4K PEs (each image has 4096 pixels). (a) Image class number 1. (b) Image class number 2. (c) Distorted input image. (d) Output image obtained after convergence.

6 simulations with different problem sizes, ranging from 32 to 1024, were run for the 1K PEs case and, 7 simulations with different problem sizes, ranging from 64 to 4096, were run for the 4K PEs case. Fig. 3.7 (a)&(b) display the two image classes, Fig. 3.3 (c) displays the distorted input image, and Fig. 3.3 (d) displays the output of the neural network model after convergence. For this test, 1024 PEs were utilized. Fig. 3.8 (a)&(b) display the two image classes, Fig. 3.8 (c) displays the distorted input image, and Fig. 3.8 (d) displays the output of the neural network model after convergence. For this test, 4096 PEs were utilized.

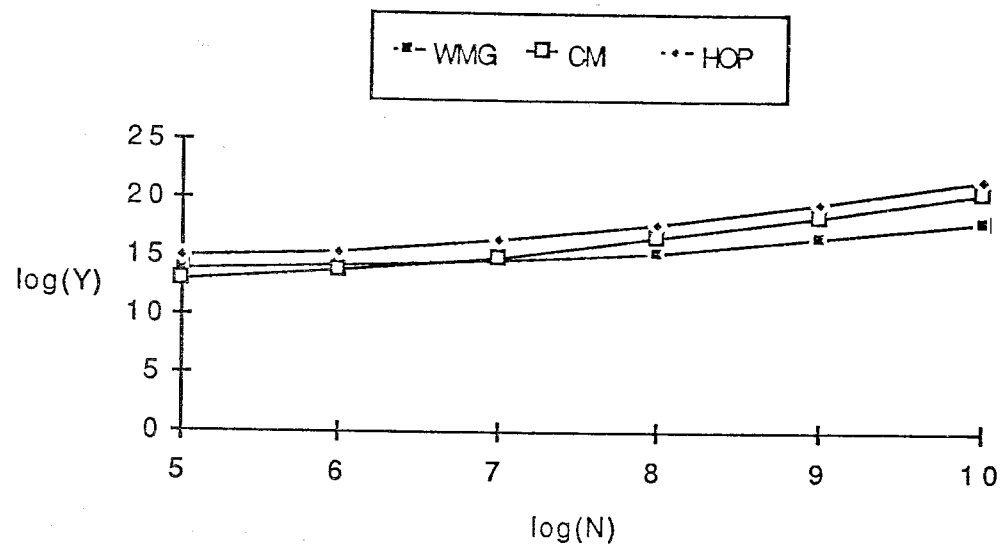
Problem Size (N)	WMG cycles	CM cycles	HOP cycles
32	16,580	8,724	34,028
64	20,290	14,451	49,192
128	28,994	35,679	100,352
256	45,260	117,231	286,000
512	117,170	436,719	990,608
1,024	330,610	1,707,136	3,744,882

(a) 1K PEs.

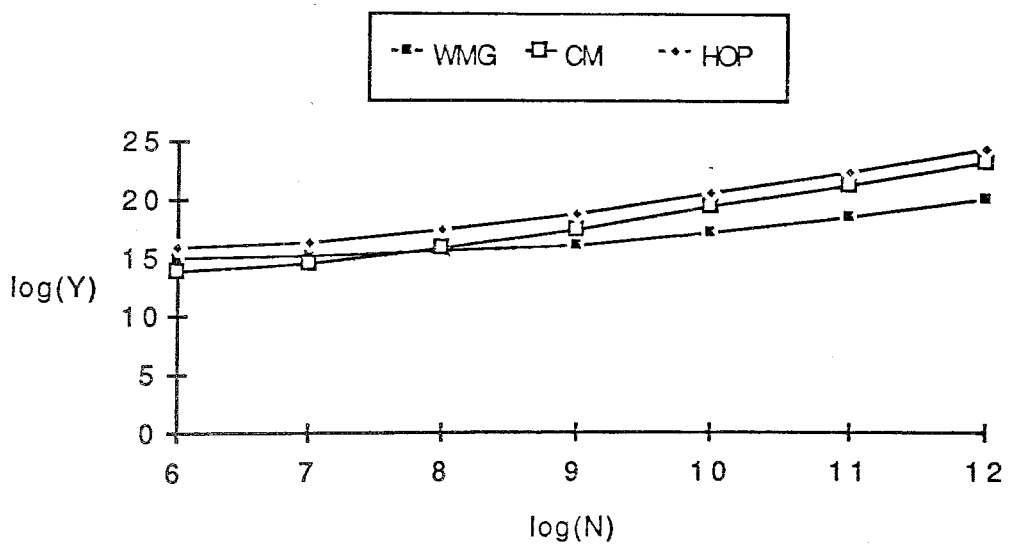
Problem Size (N)	WMG cycles	CM cycles	HOP cycles
64	32,830	16,752	66,334
128	39,484	26,851	93,186
256	54,076	64,095	182,266
512	79,174	206,767	501,930
1,024	177,580	764,847	1,707,274
2,048	438,124	2,971,951	6,382,026
4,096	1,287,916	11,765,260	24,818,436

(b) 4K PEs.

Table 3.1 The number of processor cycles for the simulations run as returned by the Blitzen simulator.



(a) 1K PEs.



(b) 4K PEs.

Fig. 3.9 The logarithm of the number of cycles (log(Y)) vs. the logarithm of the problem size (log(N)) (graph of the data given in Tables 3.1 (a) and (b), respectively).

Tables 3.1 (a) and (b) show the number of processor cycles it takes to simulate Hopfield network models of different problem sizes on 1K and 4K PEs, respectively. A row of in each table correspond to a single simulation. The first column, denoted by N, gives the problem size for that simulation. The second column, denoted as *weight matrix generation (WMG) cycles*, gives the number of cycles it takes in generating the corresponding weight matrix. The third column, denoted as *class matching (CM) cycles*, gives the number of cycles it takes in a single class matching iteration. Finally, the fourth column, denoted as *Hopfield (HOP) cycles*, gives the number of cycles it takes in the overall simulation of the Hopfield network model.

The data given in Tables 3.1 (a) and (b) are plotted in Fig. 3.9 (a) and (b), respectively, as the logarithm of the number of cycles, log(Y), vs. the logarithm of the problem size, log(N).

Results by the Cycle Estimation Functions for the BMPP: Tables 3.2 (a) and (b) lists the values taken by the cycle estimation functions for the BMPP in the cases of 16K and 64K PEs, respectively, for various problem sizes under the Hopfield network model conditions set at the beginning of Section 3.3.1. The format of Table 3.2 is the same as the format of the Table 3.1.

Problem Size (N)	WMG cycles	CM cycles	HOP cycles
128	65,592	32,748	131,088
256	78,134	51,411	180,956
512	104,502	119,967	344,436
1,024	162,374	381,999	926,372
2,048	298,662	1,405,743	3,110,148
4,096	653,414	5,451,951	11,557,316
8,192	1,691,622	21,539,250	44,770,122
16,384	5,082,854	85,693,360	176,469,574

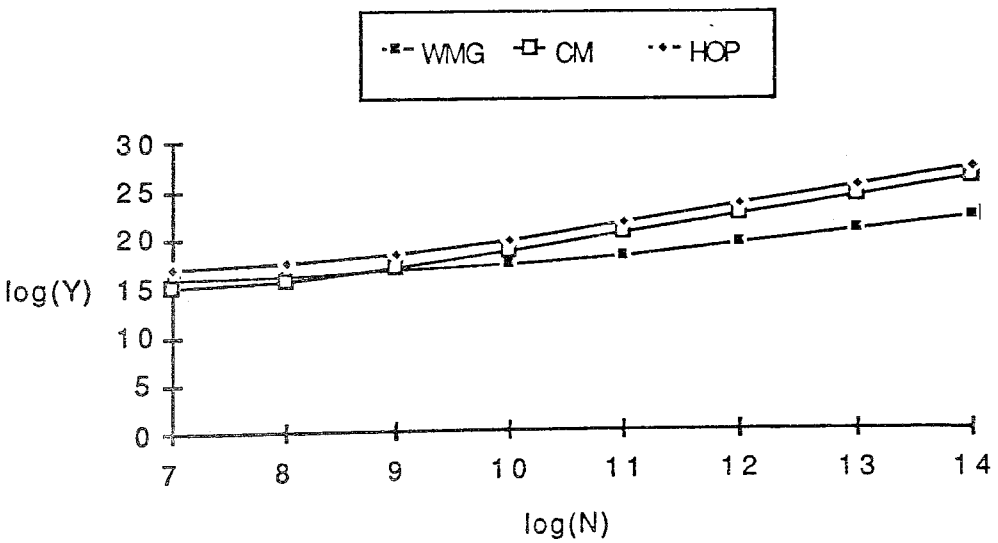
(a) 16K PEs.

Problem Size (N)	WMG cycles	CM cycles	HOP cycles
256	131,634	64,680	260,994
512	155,952	100,291	356,534
1,024	205,872	230,751	667,374
2,048	310,848	728,623	1,768,094
4,096	541,344	2,672,175	5,885,694
8,192	1,084,512	10,350,510	21,785,532
16,384	2,499,552	40,872,110	84,243,772
32,768	6,644,448	162,575,000	331,794,448
65,536	20,193,500	648,619,700	1,317,433,000

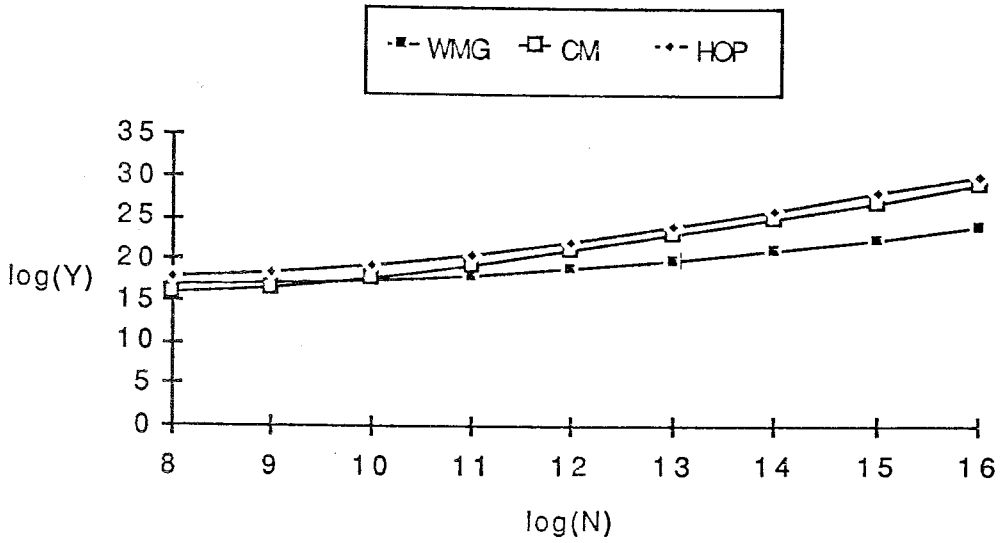
(b) 64K PEs.

Table 3.2 The number of cycles found using the cycle estimation functions for the BMPP.

The data given in Tables 3.2 (a) and (b) are plotted in Fig. 3.10 (a) and (b), respectively, as the logarithm of the number of cycles, log(Y), vs. the logarithm of the problem size, log(N).



(a) 16K PEs.



(b) 64K PEs.

Fig. 3.10 The logarithm of the number of cycles (log(Y)) vs. the logarithm of the problem size (log(N)) (graph of the data given in Tables 3.2 (a) and (b), respectively).

3.3.2 Results for the ISP

Problem Size(N)	WMG cycles	CM cycles	HOP cycles
32	16,604	37,056	90,176
64	60,800	147,840	356,480
128	236,288	590,592	1,417,472
256	931,328	2,360,832	5,652,992
512	3,697,664	9,440,256	22,578,186
1,024	14,735,360	37,589,980	90,245,120
2,048	58,830,850	151,007,200	360,845,300
4,096	235,102,200	604,004,400	1,443,111,000
8,192	939,966,500	2,415,968,000	5,771,903,000
16,384	3,758,981,000	9,663,775,000	23,086,530,000
32,768	15,034,160,000	38,654,900,000	92,343,960,000
65,536	60,133,080,000	154,619,200,000	369,371,500,000

Table 3.3 The number of cycles found using the cycle estimation functions for the ISP.

Table 3.3 lists the values taken by the cycle estimation functions for the ISP at various problem sizes under the Hopfield network model conditions set in Section 3.3.1. The format of Table 3.3 is the same as the format of Table 3.1.

The data given in Table 3.3 is plotted in Fig. 3.11, respectively, as the logarithm of the number of cycles, log(Y), vs. the logarithm of the problem size, log(N).

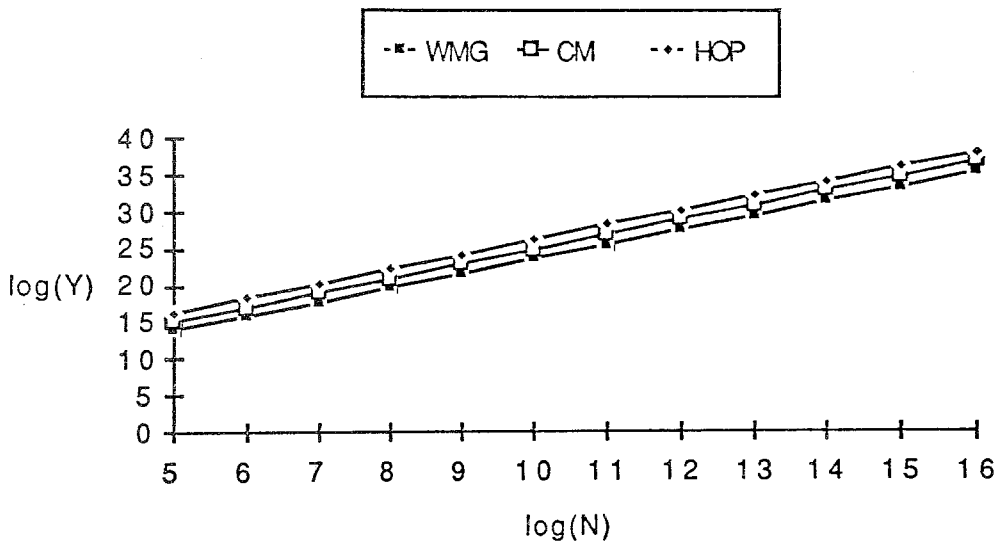


Fig. 3.11 The logarithm of the number of cycles (log(Y)) vs. the logarithm of the problem size (log(N)) (graph of the data given in Table 3.3).

3.4 Analysis of Simulation Results and Discussions

The results given in Section 3.3 are analyzed for speedup (S) and efficiency (P) which are described in Section 2.2.5.

3.4.1 Speedup Analysis

Tables 3.4 (a) and (b) give the speedups obtained using 1K and 4K processing elements (PEs) and these tables are based on the data obtained on the Blitzen simulator. Tables 3.4 (c) and (d) give the speedups obtained using 4K and 16K PEs and are based on the data obtained from the cycle estimation functions for the Blitzen massively parallel processor (BMPP). These results reflect the advantage of using the BMPP compared to the imaginary sequential processor (ISP) (see Appendix E for the ISP). Each row in the tables corresponds to a single simulation. The first column, denoted by N, gives the problem size for that simulation. The second column, denoted as *weight matrix generation (WMG) speedup*, gives the speedup obtained in generating the corresponding weight matrix. The third column, denoted as *class matching (CM) speedup*, gives the speedup obtained in a single class matching iteration. Finally, the fourth column, denoted as *Hopfield (HOP) speedup*, displays the speedup obtained in the overall simulation of the Hopfield network model.

Problem Size (N)	WMG Speedup	CM Speedup	HOP Speedup
32	1.00	4.24	2.65
64	2.99	10.23	7.24
128	8.14	16.55	14.12
256	20.57	20.13	19.76
512	31.55	21.61	22.79
1024	44.57	22.01	24.09

(a) 1K PEs.

Problem Size (N)	WMG Speedup	CM Speedup	HOP Speedup
64	1.85	8.82	5.37
128	5.98	21.99	15.21
256	17.22	36.83	31.01
512	46.70	45.65	44.98
1024	82.97	49.14	52.85
2048	134.27	50.81	56.54
4096	182.54	51.33	58.14

(b) 4K PEs.

Problem Size (N)	WMG Speedup	CM Speedup	HOP Speedup
128	3.60	18.03	10.81
256	11.91	45.92	31.23
512	35.38	78.69	65.55
1024	90.74	98.40	97.41
2048	196.98	107.42	116.02
4096	359.80	110.78	124.86
8192	555.65	112.16	128.92
16384	739.54	112.77	130.82

(c) 16K PEs.

Problem Size (N)	WMG Speedup	CM Speedup	HOP Speedup
256	7.07	36.50	21.65
512	23.71	94.12	63.32
1024	71.57	162.90	135.22
2048	189.25	207.25	204.08
4096	434.29	226.03	245.18
8192	866.71	233.41	264.94
16384	1503.86	236.43	274.04
32768	2262.66	237.76	278.31
65536	2977.84	238.38	280.37

(d) 64K PEs.

Table 3.4 Speedup values obtained with different number of processing elements (PEs).

The speedup values given in Tables 3.4 (a) to (d) are displayed in Fig. 3.12, 3.13, and 3.14 as the logarithm of the speedup (log(S)), vs. the number of PEs (P) for the WMG, CM, and HOP cases. The graphs of Fig. 3.15, 3.16, 3.17 display the same data as log(S) vs. the logarithm of the problem size (log(N)) for the WMG, CM, and HOP cases. The graphs in Fig. 3.18 (a) to (c) display the cases, where the problem size (N) is equal to the number of PEs (P).

For the simulation of the WMG, based on the equation (3.23), one expects the speedup (S) to increase and then to converge to a certain value in the order of the problem size (N) as the number of the PEs (P) is increased. In Fig. 3.12, given a certain N for the WMG, as the number of PEs is increased, we observe that S increases until a so called threshold value which will be denoted as P*. However, we also observe a decrease in S after P*. Examining the data given in Tables 3.4 (a) to (d), we find that P* is 4K for the problem size of 512 pixels and 16K for the problem size of 1024 pixels.

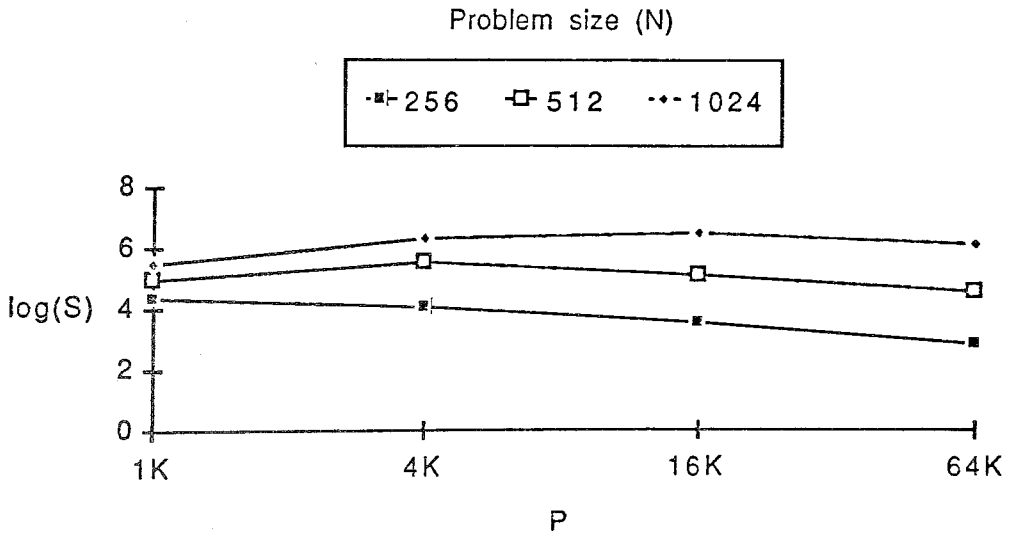


Fig. 3.12 The logarithm of the speedup (log(S)) vs. the number of PEs (P) for the WMG with different problem sizes (N).

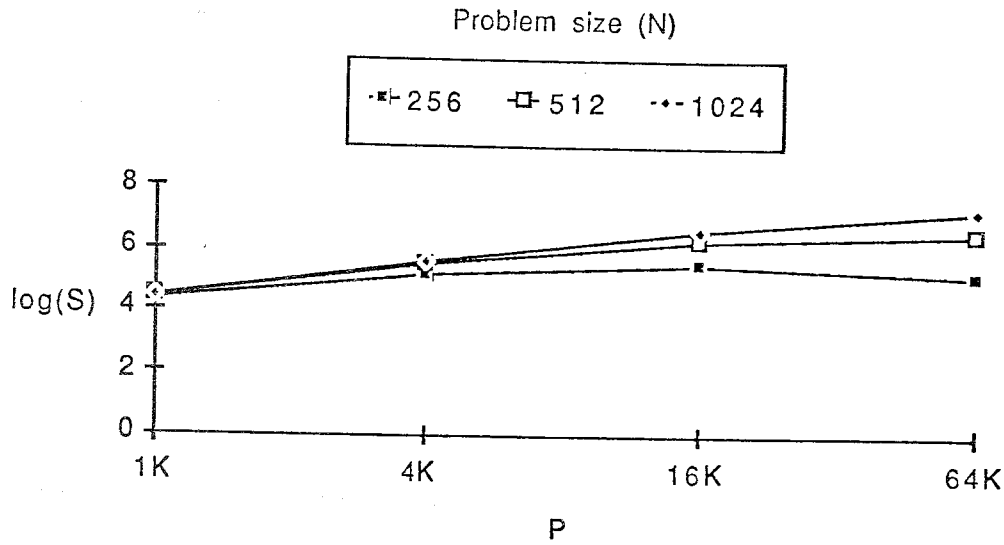


Fig. 3.13 The logarithm of the speedup (log(S)) vs. the number of PEs (P) for the CM with different problem sizes (N).

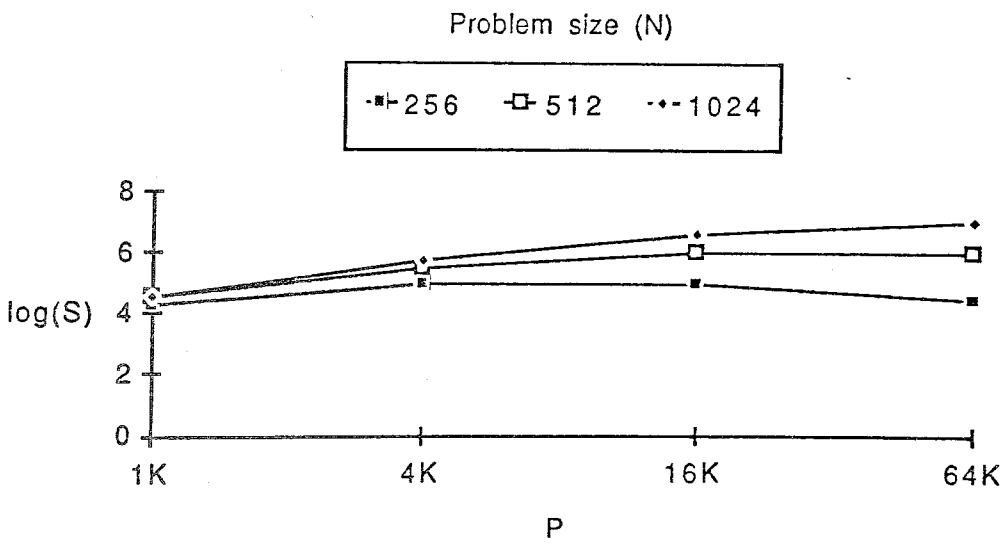


Fig. 3.14 The logarithm of the speedup (log(S)) vs. the number of PEs (P) for the HOP with different problem sizes (N).

The decrease in S occurs only at problem sizes relatively small compared to the number of PEs (P). This decrease is due to the cycles generated by the operations like the transposition of a block or indexing of PEs (see the code in Appendix C and cycle estimation functions for the BMPP in Appendix D for these operations and their contributions to the total number of cycles of a simulation). Such operations, which we call *overhead generating operations*, are required for the actual simulation of the Hopfield network model on the BMPP. They do not appear in the time complexity, speedup, or efficiency of any of the parallel algorithms used in the simulations of this study, because by generating cycles proportional to the square root of the number of PEs, they do not affect the overall time complexity of any of the parallel algorithms used in the simulations. However, for low N/P ratios, they may contribute to the number of cycles used in a simulation as much as the other major operations, and therefore, they can lower the speedup obtained after a threshold value (P*). P* is roughly estimated as $N^2/2^6$ for the simulation of the WMG on the BMPP architecture.

In Fig. 3.13, given a certain N for the CM, as P is increased, we observe that S increases until a threshold value (P*), and then S decreases. This is expected since we also use the transposition operation to simulate the CM on the BMPP. However, S continues to increase until the N/P ratio gets very small and this can be explained examining the equation (3.33), which states that, for a given N, S increases in the order of the square root of the increase in P. Examining the data given in Tables 3.4 (a) to (d), we roughly estimate P* as $N^2/2^2$ for the simulation of the CM on the BMPP architecture and this is much greater than P* of the WMG.

In Fig. 3.14, given a certain N for the HOP, as P is increased, we observe that S increases until a certain P*, and beyond P*, S decreases. The P* of the HOP is estimated approximately as $N^2/2^2$ based on the data given in Table 3.4 (a) to (d) for the simulation of the HOP on the BMPP for the Hopfield network model conditions set in Section 3.3.1. Therefore, P* of the HOP is smaller than the P* of the WMG but nearly the same as the P* of the CM.

The speedup values and graphs displayed in Tables 3.4 (a) to (d) and Fig. 3.14 for the HOP are very similar to the speedup values and graphs of the CM displayed in Tables 3.4 (a) to (d) and Fig. 3.13, respectively This is also what we have found after analyzing the parallel implementation of the HOP on the hypothetical array processor (HAP) in Section 3.2.3.

Based on the equation (3.23), one expects, for the simulation of the WMG with a certain number of PEs (P), that the speedup (S) would increase and then converge to a certain value in the order of P as the problem size (N) is increased. In Fig. 3.15, we observe that S increases as N is increased. Fig. 3.15 is not be expected to illustrate the convergence since it only displays the speedup values of the cases at which N is less than or equal to P. One must have the speedup values for the cases at which N is greater than P, as equation (3.23) shows, to be able to display the convergence. Therefore, we can not say anything about the convergence merely based on Fig. 3.15. However, we observe in Fig. 3.12 that S increases at decreasing rates as N is increased, and this is in accordance with the equation (3.23).

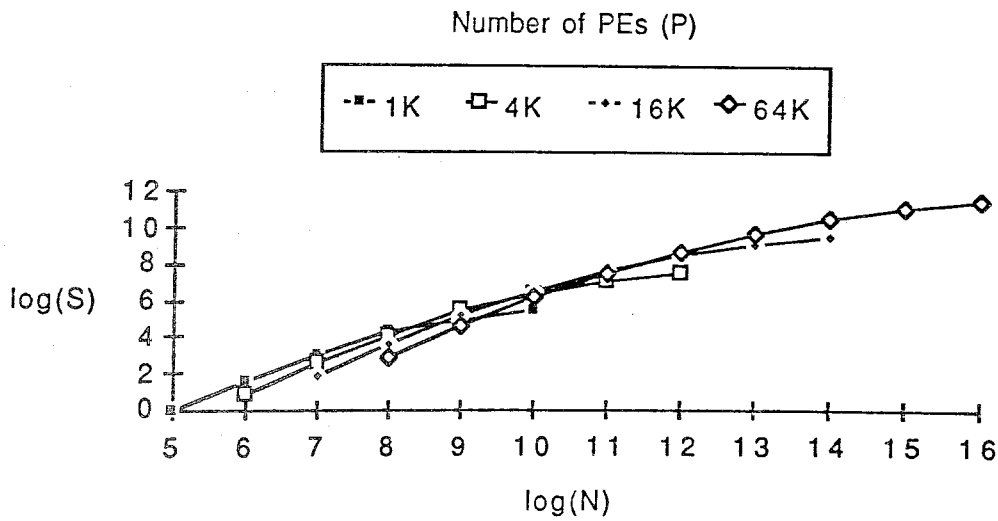


Fig. 3.15 The logarithm of the speedup (log(S)) vs. the logarithm of the problem size (log(N)) for the WMG with different number of processing elements (PEs).

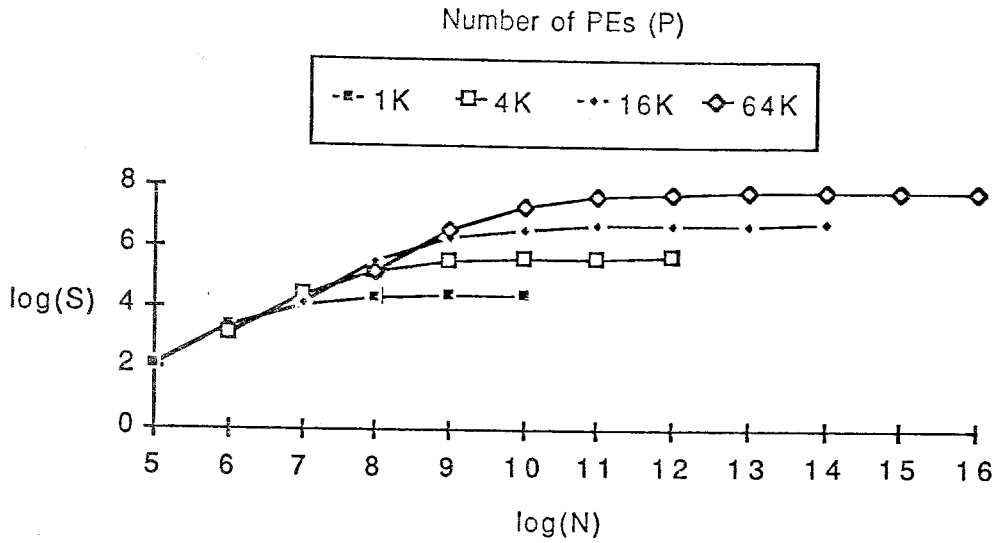


Fig. 3.16 The logarithm of the speedup (log(S)) vs. the logarithm of the problem size (log(N)) for the CM with different number of processing elements (PEs).

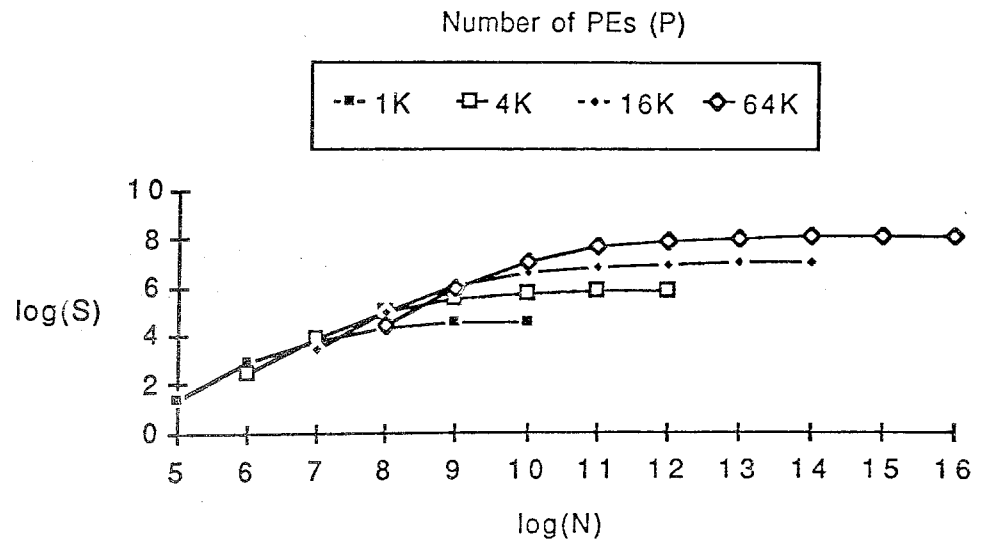


Fig. 3.17 The logarithm of the speedup (log(S)) vs. the logarithm of the problem size (log(N)) for the HOP with different number of processing elements (PEs).

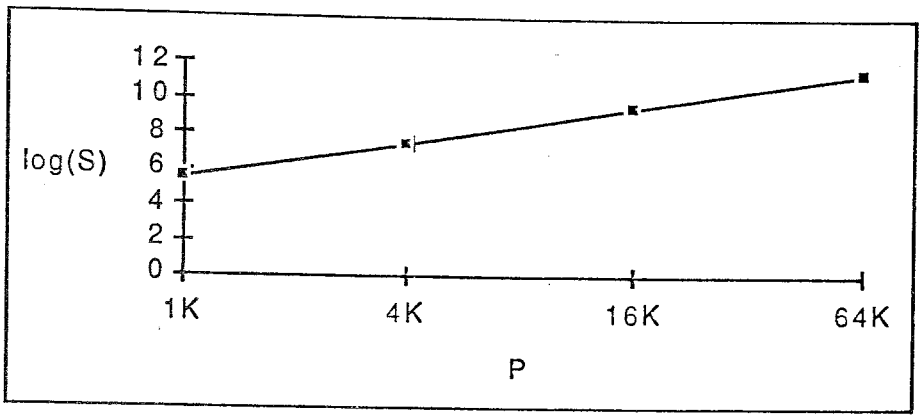
In Fig. 3.16, given a certain P for the CM, we observe that S increases and converges to a certain value as N is increased. This can be estimated using the equation (3.33) since it states that the speedup converges to a certain value in the order of the square root of P.

In Fig. 3.17, given a certain P for the HOP, as N is increased, we observe that S increases and converges to a certain value. The speedup values and graphs displayed in Tables 3.4 (a) to (d) and Fig. 3.17 for the HOP are again very similar to the speedup values and graphs of the CM displayed in Tables 3.4 (a) to (d) and Fig. 3.16, respectively

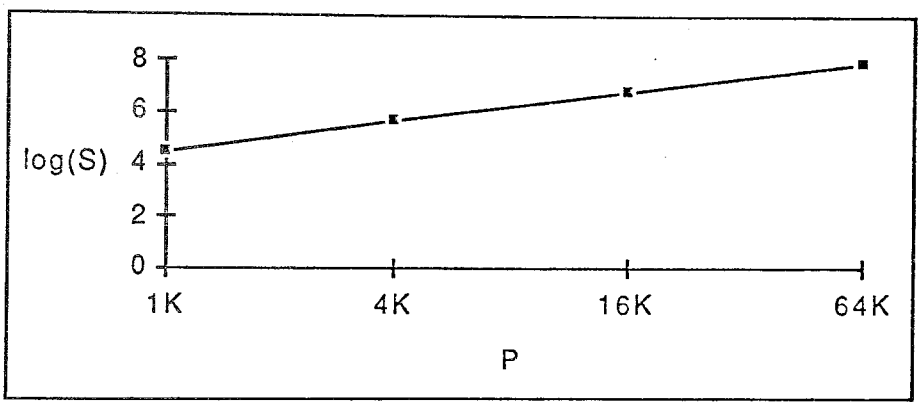
In Fig. 3.18 (a) to (c), the problem size (N) is always held equal to the number of PEs (P) as P is increased. In Fig. 3.18 (a), for the WMG, we observe that S increases as P is increased, and based on the data given in Table 3.4, we find that the increase in S is approximately in the order of the increase in P and this observation is in accordance with the equation (3.27).

In Fig. 3.18 (b), for the CM, we observe that S increases as P is increased, and based on the data given in Table 3.4, we find that the increase in S is approximately in the order of the increase in the square root of P and this is what equation (3.37) states.

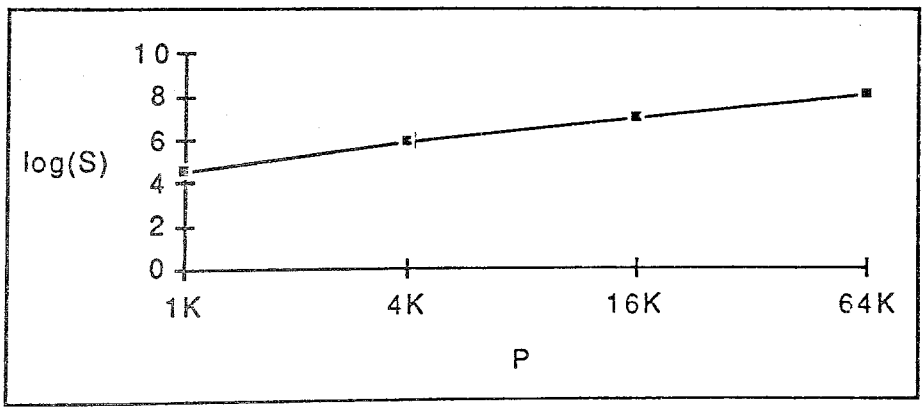
In Fig. 3.18 (c), we observe that S increases as P is increased, and based on the data given in Table 3.4, we find that the increase in S is approximately in the order of the increase in the square root of P, and this is what equation (3.47) states.



(a) WMG.



(b) CM.



(c) HOP.

Fig. 3.18 The logarithm of the speedup (log(S)) vs. the number of PEs (P). For the above cases, the problem size (N) is equal to P.

The S and P values plotted in the Fig. 3.18 (a) to (c) do not give the results ideally same as the expressions (3.27), (3.37), and (3.47), because the CM, WMG, and HOP simulations include operations with different time complexities. The transposition, indexing of PEs, and memory swapping operations to and from VRAM cause the discrepancy between the expressions and simulation results. These operations generate cycles in the order of some functions of N and/or P other than the functions of N and P which are *dominant* in cycle generation during a simulation. They contribute, in terms of cycles, less to the time complexity of the WMG, CM, or HOP compared to the operations which are dominant. Thus, as P, hence N, is increased, the increase in the number of processor cycles of a parallel simulation is slightly higher than the values expected. This causes the speedup (S) to increase slightly more than expected.

3.4.2 Efficiency Analysis

Efficiency is defined as the ratio of the speedup (S) to number of processors (P) utilized. It describes the fraction of time a PE is useful over the whole simulation. Efficiency is ideally one.

Tables 3.5 (a), (b), (c), and (d) display the efficiencies achieved using 1K, 4K, 16K, 64K processors, respectively. These results reflect the advantage of using the BMPP compared to the imaginary sequential processor (ISP) (see Appendix E for the ISP). Each row of the tables correspond to a single simulation. The first column, denoted by N, shows the problem size for that simulation. The second column, denoted as *weight matrix generation (WMG) efficiency*, gives the efficiency achieved in generating the corresponding weight matrix. The third column, denoted as *class matching (CM) efficiency*, gives the efficiency achieved in a single class matching iteration. Finally, the fourth column, denoted as *Hopfield (HOP) efficiency*, displays the efficiency achieved in the overall simulation of the Hopfield network model.

Since all the efficiency values given in Tables 3.5 (a) to (d) are smaller than one, their logarithms are negative values To prevent any confusions due to this, we use another efficiency variable, denoted as normalized efficiency (E'), in the Figures 3.19, 3.20, 3.21, 3.22, 3.23, 3.24, and 3.25 (a) to (c). E' is defined as follows,

$$E' = E*10^5. \tag{3.49}$$

The efficiency values given in Tables 3.5 (a) to (d) are displayed in Fig. 3.19, 3.20, and 3.21 as the logarithm of the normalized efficiency (log(E')) vs. the number of PEs (P) for the WMG, CM, and HOP cases. The graphs of Fig. 3.22, 3.23, and 3.24 display the same data as log(E') vs. the logarithm of the problem size (log(N)) for the WMG, CM, and HOP cases. The graphs in Fig. 3.25 (a) to (c) display the cases, where the problem size (N) is equal to the number of PEs (P).

Problem Size (N)	WMG	CM	HOP
32	0.0009	0.0041	0.0025
64	0.0029	0.0099	0.0070
128	0.0079	0.0161	0.0137
256	0.0200	0.0196	0.0193
512	0.0308	0.0211	0.0222
1024	0.0435	0.0215	0.0235

(a) 1K PEs.

Problem Size (N)	WMG	CM	HOP
64	0.0004	0.0021	0.0013
128	0.0014	0.0053	0.0037
256	0.0042	0.0089	0.0075
512	0.0110	0.0111	0.0109
1024	0.0202	0.0119	0.0129
2048	0.0327	0.0124	0.0138
4096	0.0445	0.0125	0.0141

(b) 4K PEs.

Problem Size (N)	WMG	CM	HOP
128	0.0002	0.0011	0.0006
256	0.0007	0.0028	0.0019
512	0.0021	0.0048	0.0040
1024	0.0055	0.0060	0.0059
2048	0.0120	0.0065	0.0070
4096	0.0219	0.0067	0.0076
8192	0.0339	0.0068	0.0078
16384	0.0451	0.0068	0.0079

(c) 16 K PEs.

Problem Size (N)	WMG	CM	HOP
256	0.0001	0.0005	0.0003
512	0.0003	0.0014	0.0009
1024	0.0010	0.0022	0.0020
2048	0.0028	0.0031	0.0031
4096	0.0066	0.0034	0.0037
8192	0.0132	0.0035	0.0040
16384	0.0229	0.0036	0.0041
32768	0.0345	0.0036	0.0042
65536	0.0454	0.0036	0.0042

(d) 64 K PEs

Table 3.5 Efficiencies for different problem sizes (N) with different number of processing elements (P).

For the WMG, in Fig. 3.19, we observe that the efficiency (P) decreases for a given problem size (N) as the number of PEs (P) is increased. This is in accordance with equation (3.24), which states that, for a given problem size, E decreases in the order of the increase in P.

For the CM, in Fig. 3.20, we observe that E decreases for a given N as P is increased. Comparing Fig. 3.15 (b) with Fig. 3.19, we also observe that the rate of decrease in E for the CM case is lower than the rate of the decrease in E for the WMG case. Note that these observations coincide with the equation (3.34), which states that for any problem size, E is decreased in the order of the square root of the increase in P.

For the HOP, in Fig. 3.21, we observe that E decreases, for a given N, as P is increased. Comparing Fig. 3.21 with Fig. 3.19, it is also observed that the rate of decrease in E for the HOP case, is close to the rate of the decrease in E for the CM case. This is expected by the equation (3.44) since equation (3.44) returns similar results with equation (3.34) for the problem size over the number of PEs ratios slighlty less than one or more.

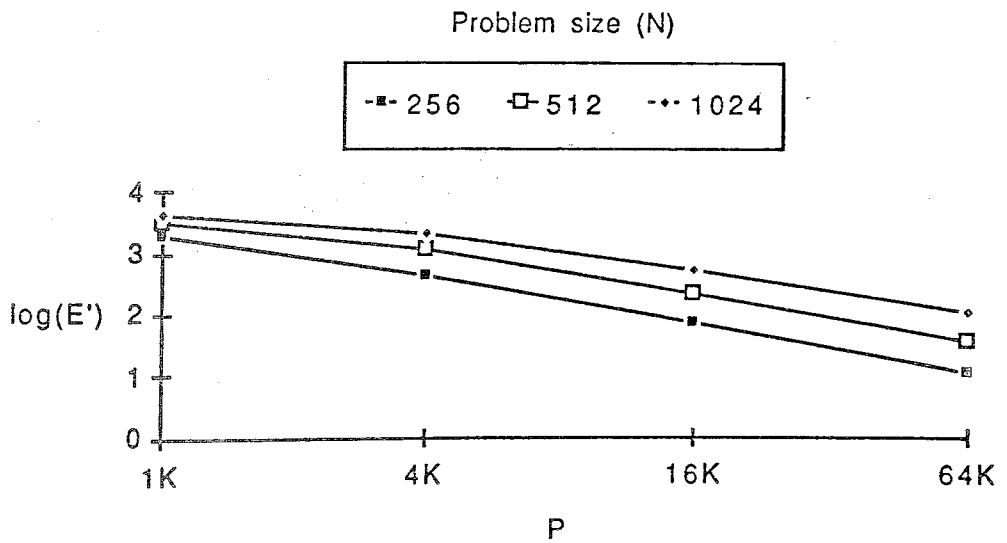


Fig. 3.19 Logarithm of the normalized efficiency (log(E')) vs. the number of PEs (P) for the WMG with different problem sizes (N).

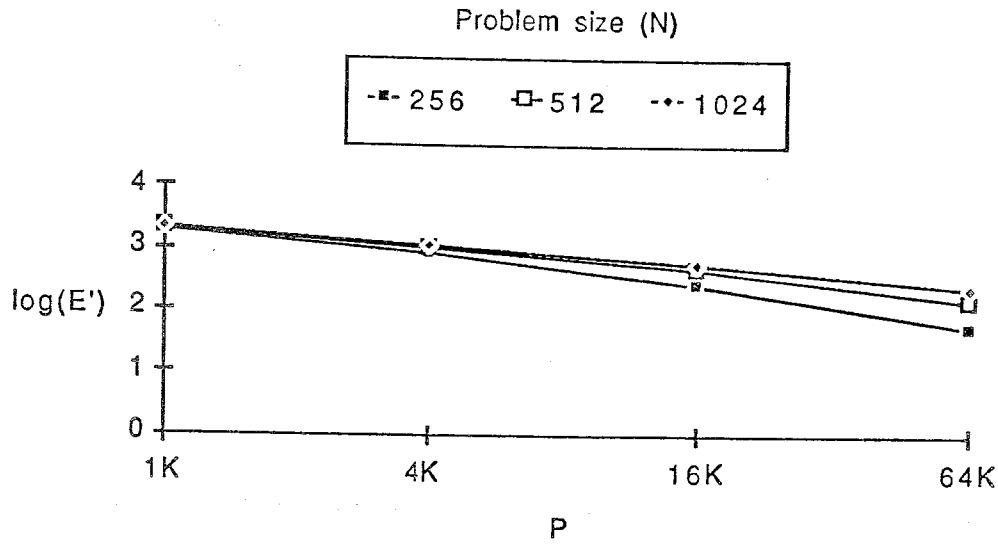


Fig. 3.20 Logarithm of the normalized efficiency (log(E')) vs. the number of PEs (P) for the CM with different problem sizes (N).

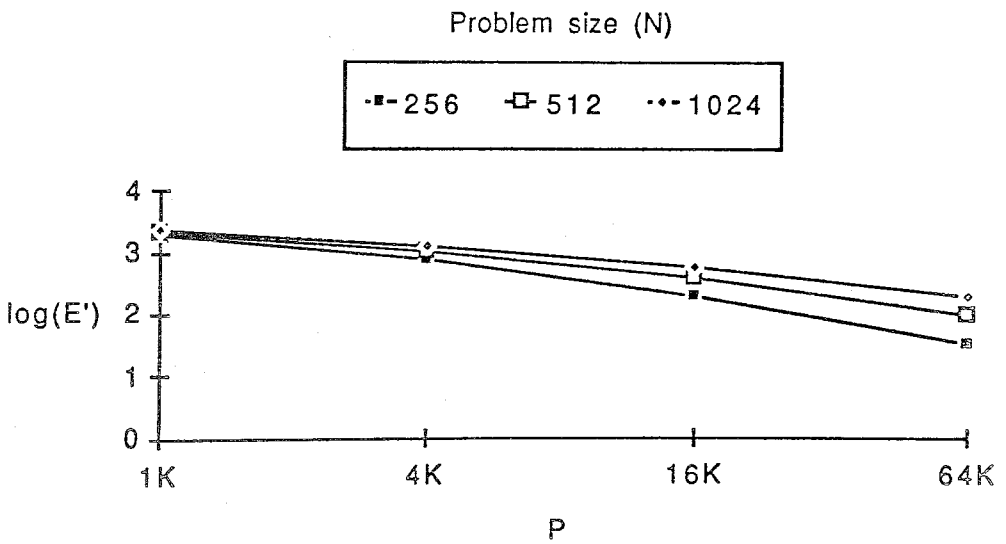


Fig. 3.21 Logarithm of the normalized efficiency (log(E')) vs. the number of PEs (P) for the HOP with different problem sizes (N).

Examining equation (3.24), one expects the efficiency (E) of the WMG to increase and then converge to a certain value as the problem size (N) is increased for a given number of PEs (P). In Fig. 3.22, it is observed that the efficiency (E) is increased as the problem size (N) is increased. However, we can observe in Fig. 3.22 that E is increased at decreasing rates as N is increased, and this is in accordance with the equation (3.24).

For the CM, in Fig. 3.23, it is observed that E increases as N is increased and E converges to a certain value for a given P. This observation agrees with the equation (3.34), which states that for a given P, E is independent of N and E is in the order of the inverse of the square root of the P.

Fig. 3.24 shows that efficiency (E) of the HOP increases and converges to a certain value as N is increased.

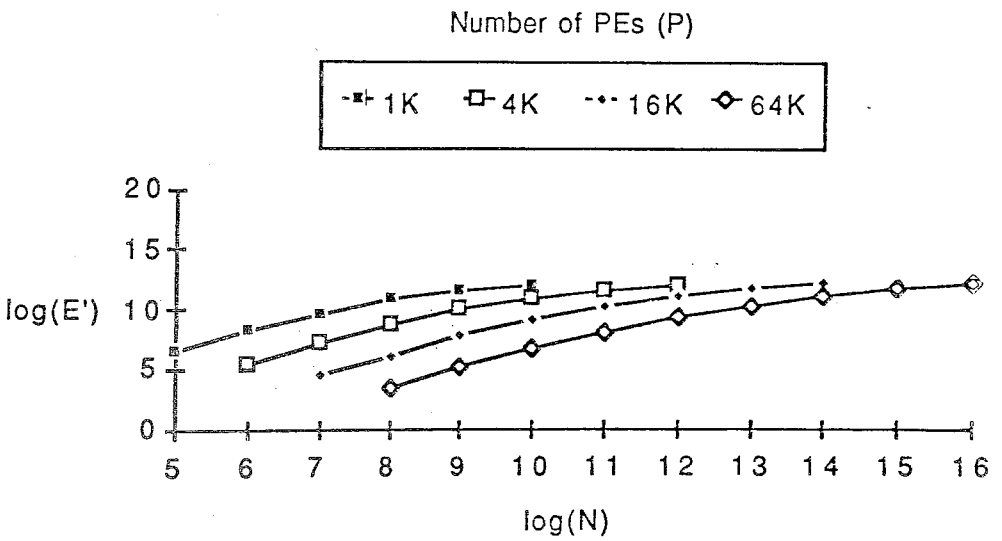


Fig. 3.22 Logarithm of the normalized efficiency (log(E')) vs. the logarithm of the problem size (log(N)) for the WMG with different number of PEs (P).

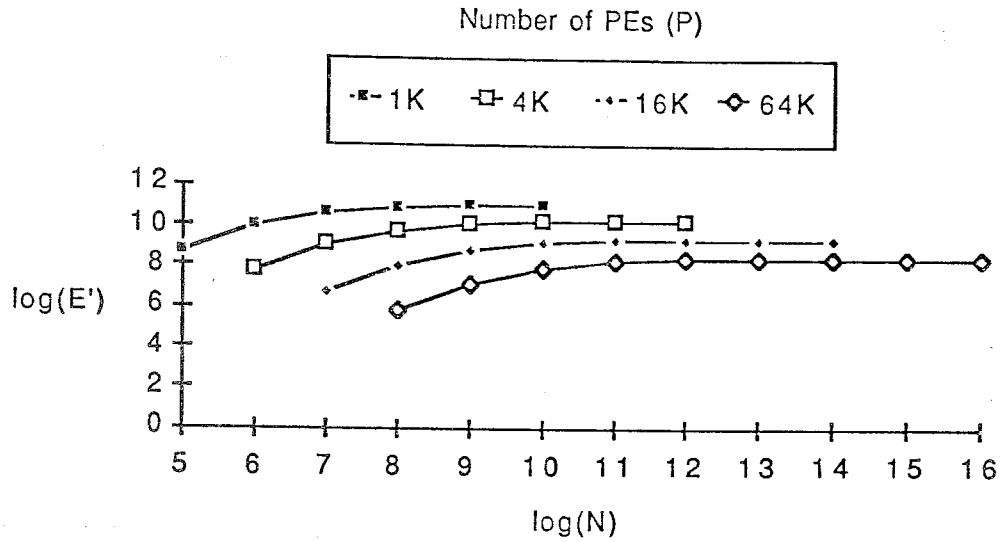


Fig. 3.23 Logarithm of the normalized efficiency (log(E')) vs. the logarithm of the problem size (log(N)) for the CM with different number of PEs (P).

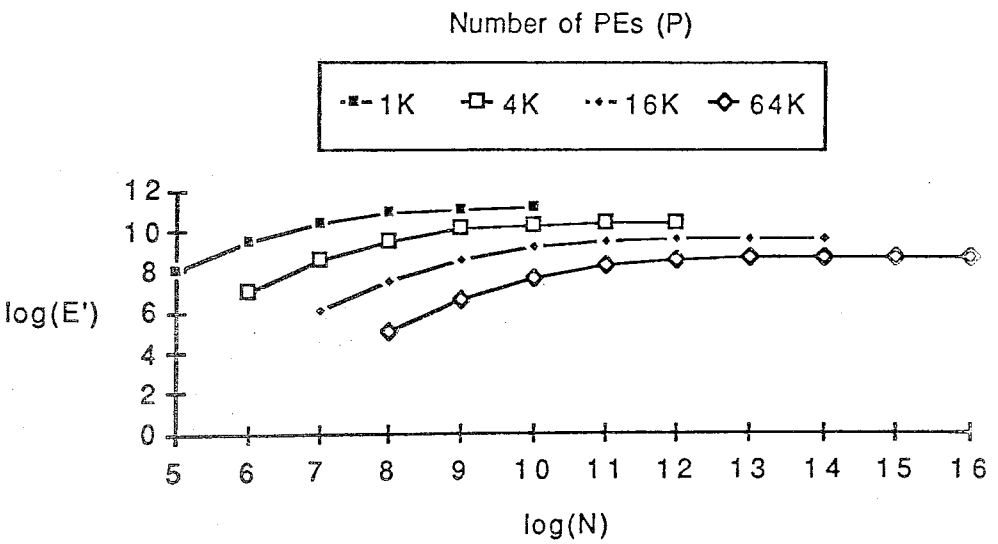
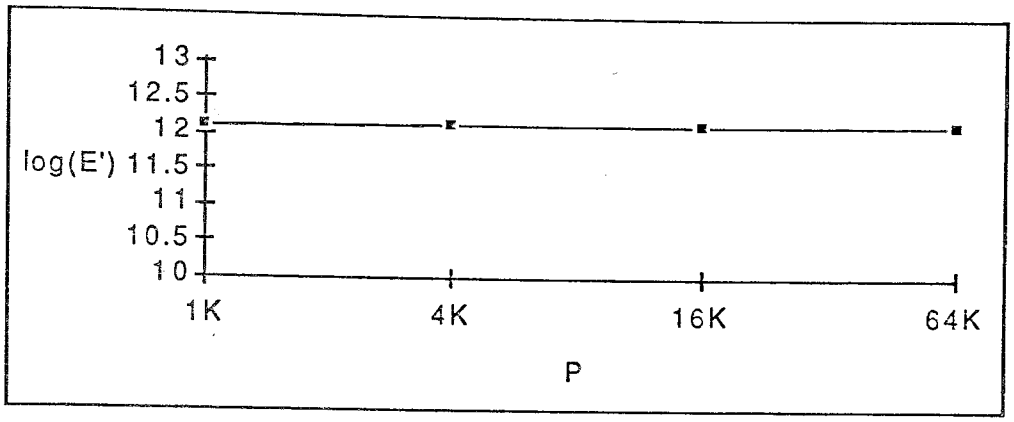
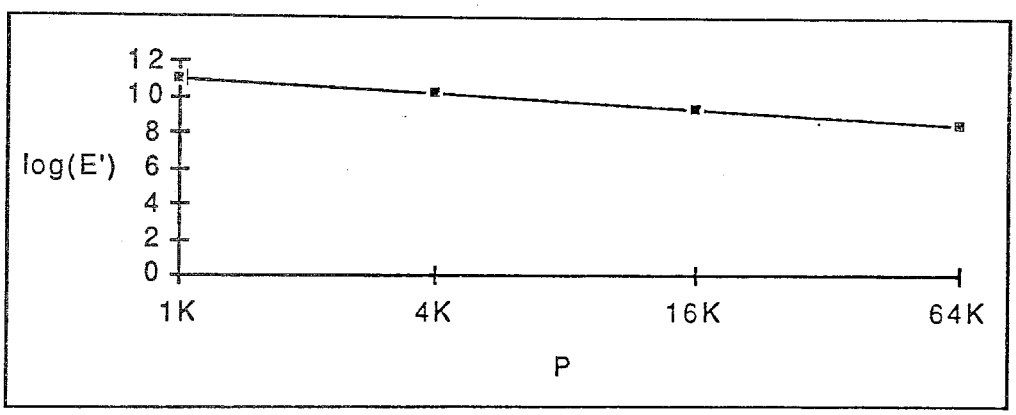


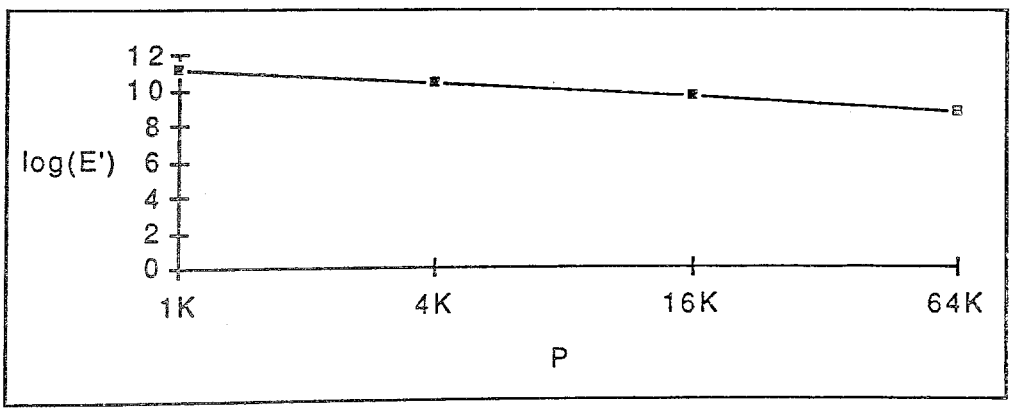
Fig. 3.24 Logarithm of the normalized efficiency (log(E')), vs. the logarithm of the problem size (log(N)) for the HOP with different number of PEs (P).



(a) WMG.



(b) CM.



(c) HOP.

Fig. 3.25 Logarithm of the normalized efficiency (log(E')) vs. the number of PEs (P) graphs for the case where the problem size (N) is equal to P.

In Fig. 3.25 (a), it is observed for the WMG that efficiency (E) remains approximately constant for a given problem size (N) at any number of PEs (P) as equation (3.38) states.

In Fig. 3.25 (b), it is observed that for the CM that E decreases as P is increased, and using tables 3.5 (a) to (d), we find that this decrease is approximately in the order of the square root of the increase in P.

In Fig. 3.25 (c), it is observed that for the HOP that E decreases as P is increased, and using Tables 3.5 (a) to (d), we find that this decrease is approximately in the order of the square root of the increase in the number of PEs.

IV CONCLUSION AND RECOMMENDATIONS

4.1 Conclusion

In this study, the main objective is to demonstrate the advantage of using a massively parallel processor for the simulation of the Hopfield network model. The simulation results are analyzed for the speedup achieved by the Blitzen massively parallel processor (BMPP) compared to an imaginary sequential processor. The processor utilization, which is measured as the efficiency of a parallel algorithm, is also studied.

After analyzing the simulation results in Section 3.4, it is observed that one achieves different speedup and efficiency values in the simulation of the weight matrix generation (WMG) part of the Hopfield algorithm, in the simulation of the class matching (CM) part of the Hopfield algorithm, and in the overall simulation of the Hopfield network model (HOP) on the BMPP architecture.

Comparison of the analytical derivations and the simulation results shows that the greater the "problem size over number of processors" ratio, the closer the orders of the speedup and efficiency values to the analytical expressions derived for the WMG, CM, and HOP simulations. The reasons are twofold for this observation.

The first reason is that while analyzing a parallel algorithm for time complexity, speedup, and efficiency, terms in the relatively low orders are neglected. However, for "small problem size over number of processors" ratios, these terms of relatively low orders contribute to the time complexity of the algorithm as much as the major terms.

The second reason is the array processor architecture used in the simulations of this study. The BMPP architecture requires the transposition of a matrix at the beginning of both the CM and WMG simulations. Transposition operation generates cycles only in the order of the square root of the number of processors, and therefore, contributes to the time complexity of these algorithms as much as the major operations for small "problem size over number of processors" ratios.

The speedup achieved in the (WMG) part of the parallel implementation of the Hopfield algorithm is a function of the problem size and the number of processors (P). For a given problem size (N) the speedup for the WMG increases until a so called threshold number of processors which is denoted as (P*). If more than P* number of processors are used, then the speedup decreases. The threshold number of processors, at which the speedup reaches a maximum, depends on the array processor architecture. For the discussed WMG implementation on the BMPP architecture, P* is estimated roughly as $N^2/64$.

For a given number of processors, the speedup for the WMG increases as the problem size is increased. The speedup is expected to converge to a certain value in the order of the number of processors for large "problem size over number of processors" ratios. This generates a speedup in the order of the number of processors for large "problem size over number of processors" ratios. Therefore, with any size and number of processors, the speedup gain for the WMG is at most in the order of the number of processors.

The processor utilization is also considered as the efficiency of a parallel algorithm. For a given problem size, the efficiency of the parallel implementation of the WMG decreases as the number of processors is increased. That is, if one increases the number of processors in the BMPP for an increased speedup at a certain problem size, he/she will be utilizing the processors less for the WMG. However, for a given number of processors, the efficiency increases and expected to converge to a certain value in the order

of one as the problem size is increased. Therefore, independent of the problem size and number of processors (for large "problem size over number of processors" ratios), the processors used in the WMG simulation are utilized approximately in the order of one.

As the simulations have shown, for a given problem size (N), the speedup of the CM increases until a threshold number of processors (P*) and decreases as the number of processors is increased beyond the P*. However, the speedup is only a function of the number of processors ideally. This shows itself in two parts of the simulation results. The first is that the estimated P*, which is roughly $N^2/4$, for the CM, is much higher than the estimated P* of the WMG, which is roughly $N^2/64$, for our implementation on the BMPP architecture. And the second is that the increase in the speedup of the CM is in the orders of the square root of the increase in the number of processors even for the simulations done using a number of processors close to the P* for a given problem size.

For a given number of processors, the speedup for the CM increases, as the problem size is increased, and then converges to a value in the order of the square root of the number of processors. In consequence, for large "problem size over number of processors" ratios, the speedup for the CM is in the order of the square root of the number of processors. Therefore, at any problem size and number of processors, the maximum speedup of the parallel implementation of the CM on the BMPP is in the order of the square root of the number of processors employed.

The efficiency of the CM decreases as the number of processors is increased for a fixed problem size. For a fixed number of processors, the efficiency increases as the problem size is increased, and converges to a certain value in the order of one over the square root of the number of processors. Therefore, for the large "problem size over number of processors" ratios, the processors are utilized in the order of one over the square root of the number of processors. Hence, if the number of processors are increased with the aim of increased speedups, processor utilization will be less.

The overall speedup for the Hopfield algorithm increases as the number of processors is increased for a given problem size (N) until a threshold number of processors which is denoted as P* and then decreases. P* is estimated roughly as $N^2/4$ for the parallel implementation of the Hopfield algorithm on the BMPP architecture. For a fixed number of processors, the speedup for the Hopfield algorithm increases and then converges to a certain value in the order of the square root of the number of processors as the problem size is increased. Therefore, for large "problem size over number of processors" ratios, the increase in speedup is in the order of the square root of the number of processors.

The overall efficiency of the Hopfield algorithm decreases as the number of processors is increased for a fixed problem size. For a fixed number of processors, the efficiency increases and converges to a certain value in the order of one over the square root of the number of processors as the problem size is increased. Therefore, for large "problem size over number of processors" ratios, the efficiency is decreased in the order of one over the square root of the number of processors.

The speedup and efficiency of the parallel implementation of the Hopfield algorithm is governed by the class matching iterations. This is more significant in the Hopfield model simulations with large problem sizes contrary to the sequential implementation of the Hopfield algorithm in which both the weight matrix generation and class matching iterations contribute the overall time complexity in the orders of the square of the problem size.

We conclude that the speedup that can be achieved by simulating the Hopfield neural network model on the BMPP is at most in the order of the square root of the number of processors employed. The processor utilization is decreased as more processors are used in a simulation. For a fixed problem size, the maximum speedup can be achieved only at a finite number of processors, which is referred to as the threshold number of processors. If the number of processors are more than the threshold number of

processors, the speedup will be less than its maximum possible value.

4.2 Future Work

Since the performance of the parallel implementation of the Hopfield algorithm on the BMPC is governed by the class matching iterations, the parallel algorithm discussed in this study is communication bounded. If another array processor topology other than the mesh topology, e.g n-cube topology, is used, then the communication cost of the parallel implementation of the Hopfield algorithm can be decreased. Such a topology can accomplish the computational part of the Hopfield algorithm at a computational complexity in the same orders with the mesh topology and, at the same time, can enable the interprocessor communications with a communication complexity in the orders lower than the mesh topology.

Another possibility can be the employment of systolic arrays for the parallel implementation of the Hopfield algorithm. In this case, however, a special systolic array architecture must be designed for each computational step of the Hopfield algorithm like the outer product, addition of matrices, resetting of the diagonal elements, matrix vector multiplication, and comparison of the input and output images.

Finally, as the advances in the VLSI technology flourish, it is not unrealistic to wait for other computing structures which will make the imitation of the human intelligence more feasible and cost effective.

APPENDIX A

Hopfield's Artificial Neural Network Algorithm

Hopfield's artificial neural network algorithm can be divided into two parts. The first part is called the *weight matrix generation,* and the second part is called the *class matching*. The equations (A.1) and (A.2) given below describes the algorithm. In these equations, C is the number of image classes, N is the number of pixels of an image, p is a pixel element of an image class.

$$W_{ij} = \begin{cases} \sum_{c=0}^{C-1} p_i^c . p_j^c & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases} \qquad \text{for } i,j = 0 \text{ to } N-1 \qquad \text{(A.1)}$$

$$x_i(t+1) = f(w_{ij} . x_i(t)) \quad \text{for } i,j = 0 \text{ to } N-1 \qquad \text{(A.2)}$$

where $f_n(y) = \begin{cases} 1 & \text{if } y \geq 0 \\ -1 & \text{if } y < 0 \end{cases}$ and $x_i(o)$ is initialized by the input image vector.

Each input to the Hopfield network is an image with N pixels. The Hopfield network model matches a test image, called *the input image,* to other images, called *image classes* which are previously encoded in the weight matrix of the network.

In the first part, a weight matrix is generated by adding the outer products of the image class vectors and resetting the diagonal elements. The second part is a series of iterations continuing till convergence. At the first step of an iteration, the input image vector is multiplied with the weight matrix. Then, the resultant vector passes through a nonlinear function, f_n, to generate the *output image* or a vector corresponding to an image. If the output image is not the same as the input image, then the output image becomes the new input image for the network, and the same steps of the second part are repeated until no change is observed in the output image.

APPENDIX B

Blitzen Massively Parallel Processor and The Blitzen Simulator

Blitzen Massively Parallel Processor:

The Blitzen massively parallel processor (BMPP) is a prototype machine built by a research grant from NASA. It was designed in Microelectronics Center of North Carolina by a group of scientists, and its architecture is heavily based on another massively parallel computer, the Massively Parallel Processor (MPP). The BMPP is actually aimed at processing images at high speeds sent from satellites in space.

The BMPP is an SIMD array processor. It has a central control unit (CCU) and thousands of simple processing units, called processing elements (PEs). The CCU receives an instruction sent from a host computer, decodes the instruction, and broadcasts it to all other PEs. All of the PEs execute this broadcast instruction simultaneously. The CCU can pack certain instructions into a single decoded instruction. The data I/O between the PEs and the main memory of a host computer is provided by a special random access memory (RAM), called video RAM (VRAM).

The PEs can communicate with each other through an X-grid interconnection network. An X-grid provides PE links in 8 (the north, east, west, south, and four diagonals) directions. The PEs can communicate in two modes called the grid and the wraparound routing modes. In the grid mode the PEs on the input edge receive zeroes. In the wraparound mode, edges are connected either to form a cylinder or a torus shape.

A PE is basically a simple arithmetic-logic unit. Each PE has 6 single bit registers (A, B, C, G, K, P), a variable length (max 32) shift register, and a 1024 bits local memory unit. A PE can perform 16 different functions (not, and, or, xor, etc.) of logic. A PE can transfer data between its registers, local memory and VRAM via its single bit and 4-b data busses. Fig. B.1 shows the functional units of a PE.

It is possible to mask a group of PEs by setting their mask registers G and using masked instructions. The complement register K can be used in conjunction with the G register to implement simple if-then-else logic. The BMPP allows local addressing since a 10-b address broadcast to all PEs can be ORed with the most significant 10 bits of the shift register. Another feature of the BMPP is that the contents of the single bit data bus of all the PEs can be ORed. This feature is useful in checking whether there is a true condition at the end of an operation in any of the PEs at a given time (for more information on the BMPP see [10] and [11]).

The Blitzen Simulator and Software Package:

The Blitzen simulator and software package (BSSP) is a package that simulates the BMPP. It is written by Fred Heaton and is available from the FTP server "mcnc.mcnc.org" under the directory "/public/blitzen" by anonymous FTP.

The BSSP consists of some C library code which provides the simulation of the BMPP, tutorials, and examples. The programming language is actually C, and the programmer uses the C functions provided by the library code for the simulation. However, these functions simulate the assembly language specific to the BMPP. This language enables the programmer to directly command the processor but it is certainly hard to program at this stage since the

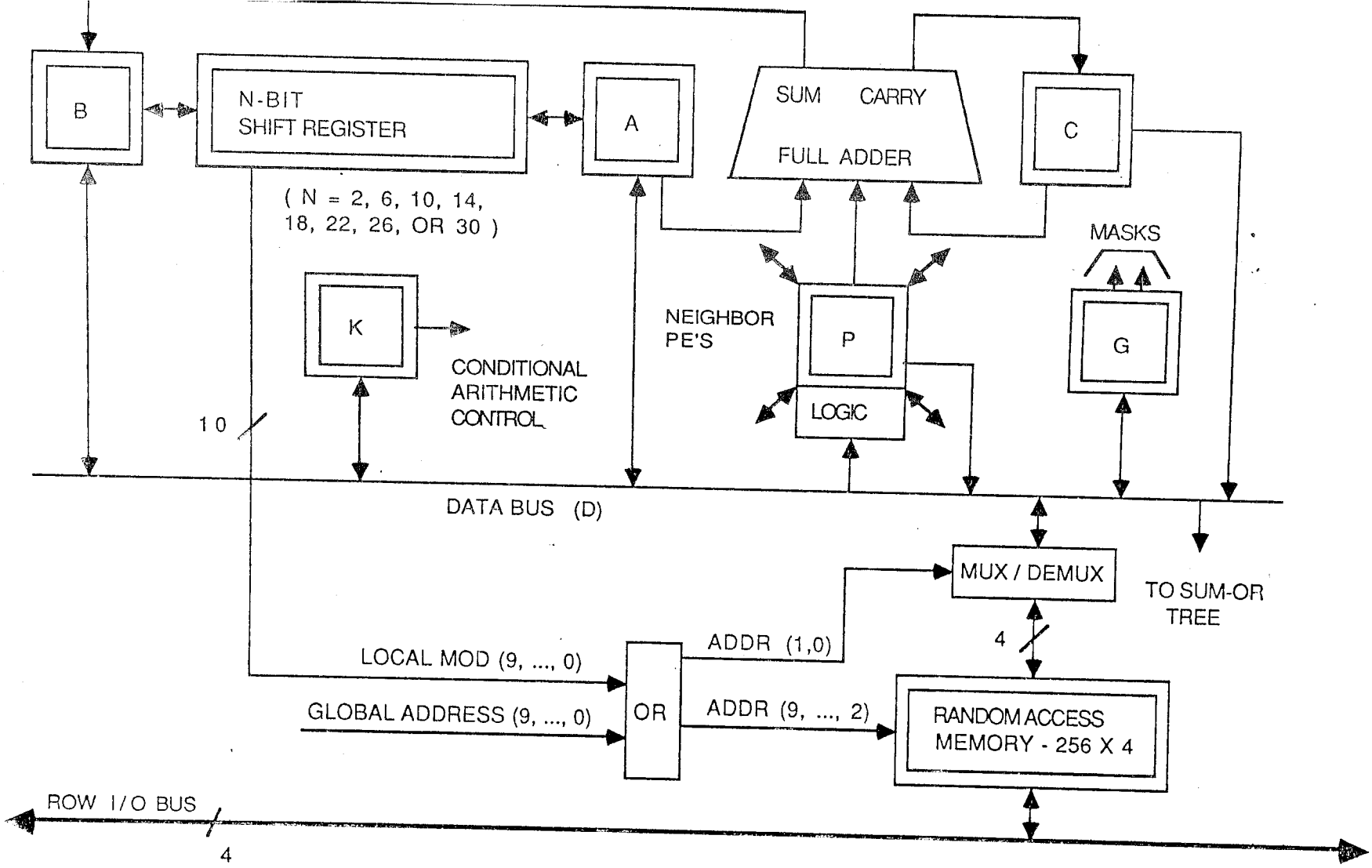


Fig. B.1 Functional parts of a BMPP processing element (PE).

assembly level instructions are bit serial. At this level, one can only make logical operations and only the addition operation on the numbers at the bit level. Therefore, the programmer must design his own Blitzen library to accomplish complex tasks like adding, subtracting, multiplying, dividing, or comparing numbers. Since the PEs are bit serial, algorithms including floating point, or fixed point operations with large bit lengths may not be efficient on the BMPP.

APPENDIX C

Simulation Code

The code that simulates the Hopfield network model is in two parts. The first part is a library, called *mylib.c* that keeps Blitzen functions accomplishing basic tasks like addition, subtraction, and multiplication between fixed point numbers, bitwise logical operations on the numbers, or more complex tasks like adding all the numbers at a row of the processor array. This library uses the Blitzen assembly level instructions provided by the Blitzen simulator. Most of this library was coded during this study and some parts of it were taken from the examples provided by the Blitzen simulator and software package. We have found the number of cycles taken by each function and included this in the explanation part of each function.

The other part, called *hopfield.c*, uses the library functions provided by mylib.c and the Blitzen simulator to simulate the Hopfield network model.The Blitzen simulator and software package along with the files mentioned above are given in the diskette.enclosed.

APPENDIX D

Cycle Estimation Functions for the Blitzen Massively Parallel Processor

Each "END" instruction in the Blitzen simulator's programming language signals the end of a processor cycle. The Blitzen simulator accomplishes the task of returning the number of cycles passed, at the end of a simulation, by counting the number of "END" instructions it encounters during the simulation. Therefore, one can also find how many processor cycles are used at the end of a simulation by counting the number of "END"s in the software code.We have used this fact to obtain the cycle data on Blitzen arrays greater than 64 by 64. An example code, which takes 2.n cycles when executed, is shown in Fig. D.1.

```
for(i=0; i<n; i++){
    MOV_MD(fromaddr+i);
    MOV_DP;
    END;

    MOV_PD;
    MOV_DM(toaddr+i);
    END;
}
```

Fig. D.1 Sample Blitzen code.

The sample Blitzen code in Fig. D.1 copies a number stored at n bits beginning at "fromaddr" to n bits beginning at "toaddr." Since there are 2 "END" instructions in the "for" loop, and since the loop turns n times, this code takes 2.n cycles when executed. Note that, since the PEs must begins and ends to execute the same instruction

at the same times, there are no branching instructions in the Blitzen simulator's language. Therefore, a PE executes all the instructions which are sent to it. Thus, the number of cycles, at a given simulation, is independent of the contents, i.e., pixel values, of the input data.

Functions, $f_{WMG,P}$, $f_{CM,P}$, $f_{HOP,P}$, are derived to estimate the number of cycles in the simulation of the weight matrix generation (WMG), a single class matching iteration (CM), and the complete Hopfield algorithm (HOP), respectively. These functions were derived after careful examination of the simulation code in Appendix C. They returned the same results as the actual simulations for 32 by 32 and 64 by 64 cases. Hence, they can be used to estimate the total number of cycles for problems with greater dimensions.

$$f_{WMG,P}(C,S,P,nw,np,nx) = C*[B^2*(4*nw + 2*np + 3) + 2*B*P^{1/2}*(5*np +3) + B*6 + 2*P^{1/2}*(5*np + 6*nx + 3) + 8*np + 5*nx + 8]+ B^2*(C-1)*4*nw + 2*((P^{1/2} - 1)*(4 + 2*dim)+nx - dim + 3) + 9*B*nw + 4*nx + 2 \quad (D.1)$$

$$f_{CM,P}(S,P,nw,np,nx) = B^2*[13*nw + 9 + nw*(1 + 5*dim + 3*P^{1/2})] + B*(8 + P^{1/2}*(3 + 5*np) + 8*nw) + 2*P^{1/2}*(3 + 6*nx + 5*np) + 23*np + 5*nx + nw +11 \quad (D.2)$$

$$f_{HOP,P}(C,R,S,P,nw,np,nx) = f_{WMG,P}(C,S,P,nw,np,nx) + R*f_{CM,P}(S,P,nw,np,n) \quad (D.3)$$

The arguments of the functions given in (D.1) to (D.3) are defined as follows, N: The number of pixels of a class or an input image (problem size), P: the number of PEs, B: $N/\sqrt{P}$, C: the number of classes, R: the number of iterations, dim: $\log(\sqrt{P})$, nw, np, nx: the number of bits used to represent a weight matrix element, an image or a class pixel, and a PE index, respectively.

The functions given in (D.1) to (D.3) are complicated for a term by term explanation. They are derived by analyzing the code given in Appendix C step by step. Here, we will only explain briefly the terms with significant contributions. One, who wonders about the other terms, must examine the code given in Appendix D. For specific terms in the below explanation, please consult to the Appendices B and C, and the description of the hypothetical array processor (HAP) in Section 2.2.1.

For (D.1), the most important terms are the ones including B^2 because B is directly proportional to the problem size. There are four operations that generate the B^2 term. These operations are described in the order they are executed. The first operation is the copying operation from VRAM prior to a multiplication and an addition (computed as $4CB^2*nw$), the second is the multiplication of the values of two pixels (computed as $3CB^2*np$), the third is the addition of a weight matrix element and the result of the previous multiplication, which are of different bit lengths, (computed as $CB^2(2*nw+np)$), and the fourth is the copying operation to VRAM (computed as $4(C-1)B^2*nw$) performed after an addition and a multiplication.

For D.2, the most significant terms are, again, the ones including B^2. The first operation, which generates cycles in the order of B^2, is the copying operation that fetches a block of the matrix to be multiplied from the VRAM and it takes $4*B^2nw$ cycles. The second is the multiplication of a vector element with a weight matrix element which takes $B^2(2*nw+4)$ cycles. The third is the addition of all the numbers at a column (the result of a sub-inner product), where each number is a PE of the column, and this takes $B^2(1+5*dim+3P^{1/2})*nw$. The fourth is the addition of the result of

the sub-inner product with the result of the previous sub-inner product, and this takes 1+3*nw cycles. The fifth is the swapping of the last addition to VRAM which takes 4*nw cycles, and the sixth is a routing operation which takes only 3 cycles to adjust mask registers of the PEs.

APPENDIX E

Cycle Estimation Functions for the Imaginary Sequential Processor

The imaginary sequential processor (ISP) has a single control unit and a single processor. This single processor is the same as one processing element (PE) of the Blitzen massively parallel processor (BMPP) except that this single processor has a large local RAM. Therefore, it takes the ISP the same number of cycles to perform the basic mathematical and logical operations as a single PE of the BMPP.

In the following, we describe the cycle estimation functions that return how many cycles it takes for the ISP pass to simulate the weight matrix generation (WMG), a single class matching iteration (CM), and the complete Hopfield algorithm (HOP).

Assume that the images presented to the Hopfield algorithm has N pixels; there are C classes, and the convergence is achieved in R iterations; nw and np are the number of bits, which are used to represent a weight matrix element and an image pixel, respectively

The ISP needs to perform CN^2 multiplications, $(C-1)N^2$ additions, and N reset operations ,i.e., to make an element zero, to generate the weight matrix. The corresponding multiplication, addition, and reset operations take 3+np, 2*nw+np, 2+nw+np cycles on the BMPC, respectively. Adding these figures after multiplying them with the number of times they are performed, the $f_{WMG,S}$ function in (E.1) can be found.

The ISP needs to perform N^2 multiplications and N(N-1) additions to perform an N-tupled vector and an N by N matrix multiplications. Each of the corresponding multiplication and addition operations take 2*nw+4 and 3*nw cycles, respectively. To complete a single class matching iteration, the ISP also needs to

perform N applications of the nonlinear function to each element of the N-tupled resultant vector, each application costing 1+np cycles, and N comparisons to compare the elements of the N-tupled input and output vectors, each comparison costing 2*np+1 cycles. Adding all these together after multiplying them with the number of times they are performed, one can obtain the function, $f_{CM,S}$, of (E.2).

Adding the functions (E.1) and (E.2), one can find the function, $f_{HOP,S}$, in (E.3) which gives the total number of cycles, which will be passed, for a whole simulation of the Hopfield network. In (E.1) to (E.3), the following notation is used, N: the problem size, C: the number of classes, R: the number of iterations, nw, np, nx: the number of bits used to represent a weight matrix element, a class or an image pixel, and a PE index, respectively.

$$f_{WMG,S}(C,S,nw,np,nx) = CN^2(3np) + (C-1)N^2(2*nw+np) + N(2+nw+np) \qquad (E.1)$$

$$f_{CM,S}(S,nw,np,nx) = N^2(2*nw+4) + N(N-1)3*nw + N.(1+np) + N(2*np+1) \qquad (E.2)$$

$$f_{HOP,S}(C,R,S,nw,np,nx) = f_{WMG,S}(C,S,nw,np,nx) + R*f_{CM,S}(S,nw,np,nx) \qquad (E.3)$$

BIBLIOGRAPHY

[1] Hwang, Kai and Briggs, A. Faye. Computer Architecture and Parallel Processing. McGraw-Hill,1984.

[2] Maresca, Massimo, "Scanning the Issue," Proceedings of IEEE, Vol. 79, No. 74, pp. 395-401, 1991.

[3] Tucker, W. Lewis and Robertson, G. G., "Architecture and Applications of the Connection Machine," Computer, 1983.

[4] Bertsetekas, D. P. and Tsitsiklis, J. N. Parallel and Distributed Computation: Numerical Methods. Prentice-Hall, 1989.

[5] Potter, J. L., "Image Processing on the Massively Parallel Computer," Computer, pp. 62-67, 1983.

[6] Potter, J. L. and Meilander, W. C., "Array Processor Supercomputers," Proceedings of the IEEE, Vol. 77, No. 12, pp. 1896-1913, 1989.

[7] Batcher, K. E., "Bit Serial Parallel Processing Systems, "IEEE Transactions on Computers, Vol. C31, No. 5, pp. 377-384, 1982.

[8] Akl, Selim G. The Design and Analysis of Parallel Algorithms. Prentice-Hall, 1989.

[9] Chen, S. and Cowan, C. F. N., Billings, S. A., and Grant, P. M., " Parallel Recursive Prediction Error Algorithm for Training Layered Neural Networks," International Journal of Control, Vol. 51, No. 6, pp. 1215-1228, 1990.

[10] Blevins, D. W., Davis, E. W., and Reif, J. H., "Processing Element and Custom Chip Architecture for the BLITZEN Massively Parallel Processor," Technical Report, Microelectronics Center of North Carolina.

[11] Heaton, R., Blevins, D., Davis, E., "A Bit Serial VLSI Array Processing Chip for Image Processing," Vol. 25, No. 2, pp. 364-367, 1989.

REFERENCES NOT CITED

Batcher, K. E., "Design of a Massively Parallel Computer," IEEE Transactions on Computing, Vol. C29, No. 9, 1980.

Crichlow, J. N. An Introduction to Parallel and Distributed Computing. Prentice-Hall, 1988.

Diederich, Joachim. Artificial Neural Networks: Concept Learning. IEEE Computer Society on Neural Networks Technology Series, 1990.

Gelenbe, Erol. Multiprocessor Performance. John Wiley&Sons, 1989.

Jain, Anil K. Fundamentals of Digital Image Processing. Prentice-Hall, 1989.

Kernighan, B. W. and Ritchie, D. M. The C Programming Language. 2nd ed. Prentice-Hall, 1988.

Lakshimivarahan, S. and Dhall, S. K. Analysis and Design of Parallel Algorithms. McGraw-Hill, 1990.

Mano, Moris. Computer System Architecture. 2nd ed. Prentice-Hall, 1982.

McClelland, J. L. and Rumelhart, D. E. Parallel Distributed Processing/Explorations in the Microstructure of Cognition, Volume 1: Foundations. MA: MIT Press, 1986.

McClelland, J. L. and Rumelhart, D. E. Parallel Distributed Processing/A Handbook of Models, Programs and Exercises. MA: MIT Press, 1987.

Rosenfeld, Azriel and Avinash C. Kak. Digital Picture Processing. Academic Press, 1976.