The OMP Supercomputer: A New RISC Multiprocessor Using Orthogonal Memory and Spanning Buses*

Kai Hwang, Dhabaleswar Panda, Santosh Rao,
Chien-Ming Cheng, Sharad Mehrotra,
and Hemarajan Nair

Technical Report No. CENG 89-31
Laboratory for Parallel and Distributed Computing

November 20, 1989

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-0781, USA

*All rights reserved. This research was supported by United States National Science Foundation under Grant No. 89-04172. For all correspondence, contact Kai Hwang at the above address or send Email via kaihwang@panda.usc.edu or use FAX (213) 745-7284.

The OMP Supercomputer: A New RISC Multiprocessor Using Orthogonal Memory and Spanning Buses*

Abstract: An *Orthogonal MultiProcessor* (OMP) supercomputer is currently under construction at the University of Southern California with research funding from NSF and assisted by several industrial partners. The prototype OMP is being built with 16 Intel i860 RISC microprocessors using custom-designed spanning buses and 256 parallel memory modules, which are 2-D interleaved and orthogonally accessed without conflicts. This paper presents the architecture design, simulation results, and the embeddings of other parallel architectures, such as *hypercube*, *mesh*, and *pyramid* onto the OMP structure. The embedding capability demonstrates the adaptability of OMP for either MIMD or SIMD operations. The OMP architecture design has been validated by a CSIM-based multiprocessor simulator developed at USC. Simulated performance data are reported for sorting and matrix algorithms. These results demonstrate scalable performance as the machine size increases from 1 to 64 RISC processors. The 16-processor OMP prototype is targeted to achieve 300 MIPS, which has been verified by a multiprocessor bandwidth analysis based on the design parameters used. MIPS rate of a simulated OMP is measured around 230 MIPS based on the sorting and matrix simulation experiments conducted. The simulation results demonstrated a 76% efficiency from the target performance. Both analytical and simulation results being reported are based on worst-case design parameters. This leaves plenty of room for further improvements in performance, as the project advances.

Contents .... **Page**

*Manuscript submitted to the 17th *International Symposium on Computer Architecture,* to be held in Seattle, Washington, May 26-31, 1990. All rights are reserved by the six coauthors on Nov. 20, 1989.

1. Introduction

The appearance of high-speed RISC processors on monolithic CMOS chips [36] and the enhancement of performance by parallel processing [18] have created a significant impact on computer industry. In this paper, we report the design and research experience of an innovative *Orthogonal Multiprocessor* (OMP) supercomputer using state-of-the-art 64-bit RISC microprocessors, a conflict-free multi-memory organization, and multi-dimensional spanning buses. This OMP architecture concept was originally conceived at the University of Southern California [20] and at Princeton University [31] during the last four years. In an earlier effort, the EMPRESS Project at ETH, Switzerland has explored a slightly different idea which was based upon an obsolete technology [10]. Presently, a 16-processor OMP prototype system is under construction at USC using the Intel i860 chips with funding from the NSF/MIPS Experimental Systems Program and technical assistances from several industrial partners.

The architectural development of OMP system has been greatly influenced by design and implementation experiences reported in the PASM [35], the NYU Ultracomputer [15], the Cedar multiprocessor [41], the Warp systolic computer [4], the Wisconsin multicube [14], and the P-RISC [27] systems. In the memory area, we choose private caches [7, 8] in a 2-level structure [5]. Previous works reported in [11], [21], and [29], have also impacted our design choices.

Through hardware prototyping and performance simulation, we show design innovations in the dual-processor i860 boards, orthogonal memory organization, and spanning buses on a large backplane board. We have joined Intel and Alliant in expanding the PAX (Parallel Architecture Extended) software standards for i860-based systems [3]. The OMP architectural strength is highlighted by its capability to embed a large class of interesting parallel architectures, such as *mesh* and *hypercube* for SIMD and the *pyramid* for MIMD applications involving large-scale matrix manipulations. These studies on architecture embeddings prove the effectiveness of a new paradigm for *orthogonal vector communication* using parallel memory modules which are pairwise-shared by multiple processors.

This article also shows how to generalize the OMP model to support massive parallelism using higher dimensional orthogonal structure of memory and spanning buses. This generalizes the works by Wittie [40], Bhuyan and Agrawal [6], and Winsor and Mudge [39]. We are using a C-language based multiprocessor simulator, locally developed at USC with the support from the *CSIM package* developed by Schwetman at MCC [32], to validate the design decisions. The OMP simulator, closely reflecting the hardware behavior, is also used to evaluate the performance of parallel algorithms developed for OMP. Simulated performance results are reported for matrix multiplication and large array sorting on the simulated OMP with machine sizes ranging from 1 to 64 RISC processors. These simulation results verified the linear scalability in performance, as machine size increases.

The OMP prototype is designed to achieve a peak performance of 300 MIPS for the 16-processor prototype. Besides hardware prototyping, we are also modifying the Mach/OS, ported to a SUN/4 host workstation, for supporting orthogonal multiprocessing using partially shared memory. We have extended the language C to *Ortho-C* for more efficient compilation and resource allocation towards vectorization and parallelization using the interleaved orthogonal memories. The initial application phase of the OMP project includes early vision processing and neural network simulations. Parallel language and algorithm developments for applications of the OMP system were reported [16]. The results presented in this paper are based upon our initial design specifications, theoretical architecture embeddings, and simulated performance data. Some of the design parameters are subject to further refinement in the remaining phases of the project.

2. Orthogonal Multiprocessor Architecture

A generalized model for orthogonal multiprocessors is presented. This model treats the 2-dimensional OMP structures proposed in [10, 20, 31] as special cases. The model also extends the hypercube generalizations reported in [6, 40]. The central idea of orthogonal multiprocessing lies in the use of multiple spanning buses to achieve conflict-free access of parallel memories. Thus, full memory bandwidth can be delivered to match with the combined bandwidth of multiple RISC processors.

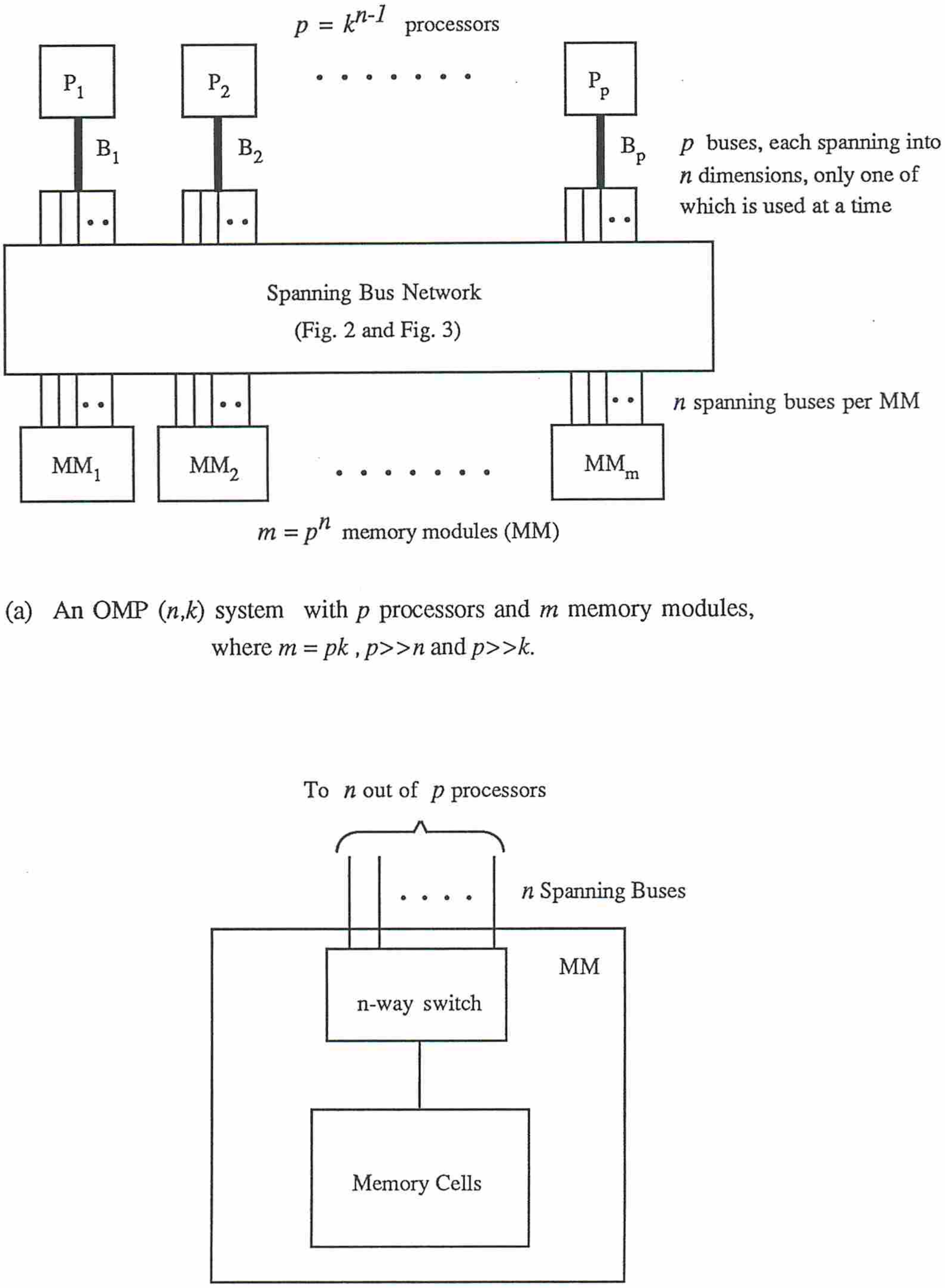
A *generalized orthogonal multiprocessor*, OMP(n, k), is characterized by two parameters, namely the *dimension* n and the *multiplicity* k as depicted in Fig.1a. There are $p = k^{n-1}$ processors and $m = k^n$ *memory modules* (MM) in the system, where $p \gg n$ and $p \gg k$. The number of MMs is k times greater than that of processors. When $k = 2$, the system is called *binary* and, in general, it is called k-ary. The system uses p memory buses, each spanning into n dimensions, but only one dimension will be used at a time. There are k MMs attached to each spanning subbus.

Each MM is connected to n out of p buses through a n-way switch, as shown in Fig.1b. It should be noted that the dimension n corresponds to the number of accessible ports that each MM has. This implies that each MM is shared by n (out of $p = k^{n-1}$) processors. In the three-dimensional case ($n = 3$) as shown in Fig.2, each MM is shared by only three processors. For example, the MM2 is shared by processors a, b, c, and d.

The orthogonal structure of an OMP(n, k) is illustrated in Fig.2 for the case of $n = 3$ and $k = 2$. Each bus is used by a single processor without time sharing and thus avoids contention. Physically, all the spanning buses from the same processor form the same bus. Logically, they are used in a mutually exclusive manner. Only MMs attached to the same spanning bus can be used by the same processor at the same time. In fact, this requirement enforces the orthogonal memory access, which results in no conflicts.

In Fig.2, we distinguish between three types of functional modules, namely the *circles* for memory modules, the *squares* for processor modules, and the *circle-inside-square* for a *computer module*. Each computer module has a local MM, which is private. The pure MMs are pairwise-shared by the processor modules. The collection of all MMs is called the *orthogonal memory*.

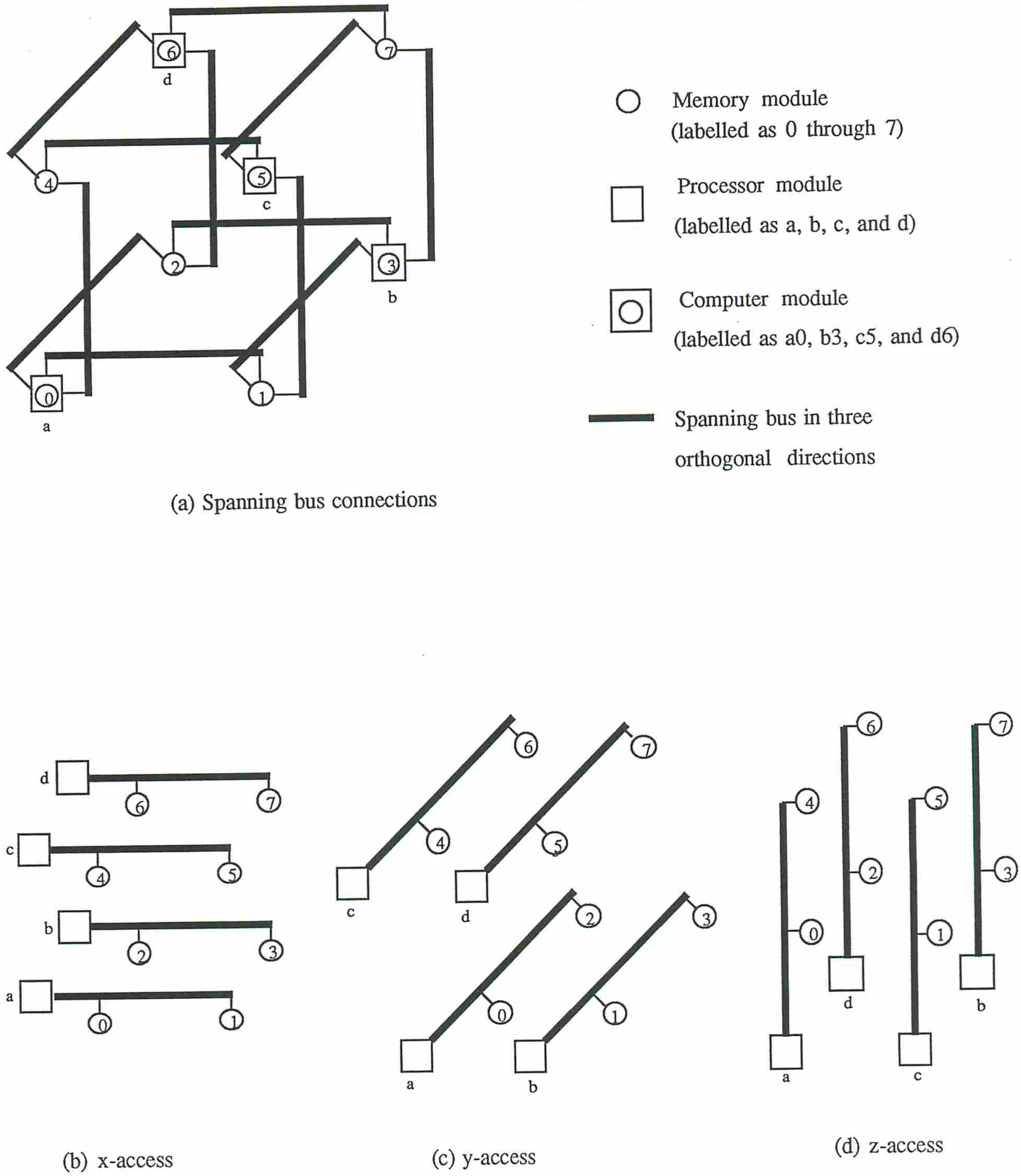
In Fig.2, four processors orthogonically access eight memory modules via four buses, each spanning into 3 directions, called the *x-access, y-access, and z-access*, respectively. It should be noted that the MMs of an OMP(n, k) form an k-ary *n-cube*, in which all the nodes


$p = k^{n-1}$ processors
P_1
P_2
P_p
B_1
B_2
B_p
p buses, each spanning into n dimensions, only one of which is used at a time
Spanning Bus Network
(Fig. 2 and Fig. 3)
n spanning buses per MM
MM_1
MM_2
MM_m
$m = p^n$ memory modules (MM)
To n out of p processors
n Spanning Buses
MM
n-way switch
Memory Cells

(a) An OMP (n,k) system with p processors and m memory modules, where $m = pk$, $p>>n$ and $p>>k$.

(b) Each memory module is connected to n processors via n spanning buses.

Fig. 1 The orthogonal multiprocessor, OMP (n,k), with an n-dimensional spanning bus network and k memory modules attached to each spanning bus.



(a) Spanning bus connections

(b) x-access

(c) y-access

(d) z-access

Fig. 2 The architecture and memory access modes of an OMP (3,2) system with 4 processors and 8 memory modules.

Table 1: Orthogonal Multiprocessor System Sizes as a Function of Dimension n and Multiplicity k.

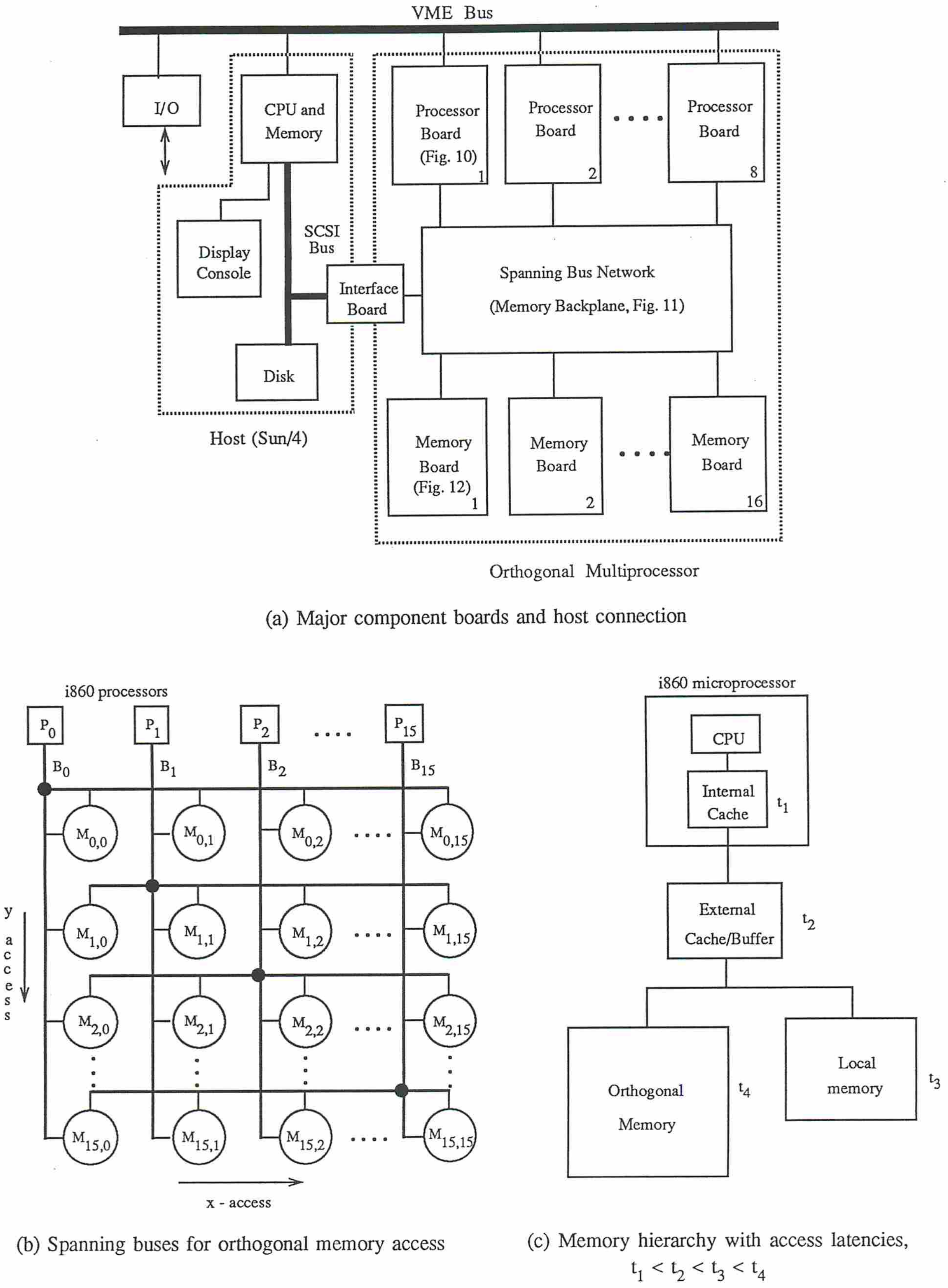
OMP(n, k)	No. of Processors $p = k^{n-1}$	No. of Memory Modules $m = k^n$
OMP(2,8)	8	64
OMP(2,16)	16	256
OMP(3,8)	64	512
OMP(3,16)	256	4,096
OMP(4,8)	512	4,096
OMP(4,16)	4,096	32,768
OMP(5,16)	65,536	524,288

are interconnected by spanning busses, instead by point-to-point links as in a hypercube architecture. Readers should not confuse the OMP(n, k) architecture with the k-ary hypercube architecture, in which all nodes are computer nodes.

For comparison purposes, various sizes of OMP architecture are listed in Table 1. A 5-dimensional OMP with multiplicity $k = 16$ will have 65K processors comparable to the connection machine. In other words, the generalized OMP can easily support massive parallelism [19], although we are building only a small prototype system with 16 processors just to prove the new architecture concept and the high speed multiprocessor technology used.

Next, we present the architecture of the prototype 2-dimensional OMP system being built at USC. Higher dimensional OMP can be built with the same *orthogonality principle* as outlined above. Based on today's technology, one can easily attach $k = 16$ MMs to each spanning bus and the dimension can be increased to $n = 5$ or higher, which is limited by packaging, performance, and cost-effectiveness considerations.

The OMP prototype system is built as a *backend* machine attached to a SUN/4 workstation *host* as shown in Fig.3. The host handles all input and output functions with the outside world. A video camera will be used to input imagery data and visual display for image output. Special operating system software supports are necessary in the host



(a) Major component boards and host connection

(b) Spanning buses for orthogonal memory access

(c) Memory hierarchy with access latencies, $t_1 < t_2 < t_3 < t_4$

Fig. 3 The prototype orthogonal multiprocessor, OMP(2,16)

to perform the following functions: *program compilation* (parallelization or vectorization besides optimization), *download* of *compiled programs* to the OMP backend, *handling of data movement* between the orthogonal memory and the mass storage in the host, *control and allocation* of OMP resources, and *performance monitoring* of the entire system. We are modifying the Mach/OS for these purposes [30]. This includes the development of a *parallelizing cross-compiler* for the extended C language, *Ortho-C*, being specially developed for orthogonal multiprocessing. A preliminary version of Ortho-C has been given in [16]. Programmer can only interact with the OMP system through the host.

The OMP machine layout consists of essentially 4 types of printed-circuit boards; namely the *processor boards* (PB), *memory boards* (MB), *spanning bus network* (SBN), and the *interface board* (IB). The IB interfaces MMs with the SCSI bus of the SUN/4 host for data uploading and downloading using DMA. Each PB houses two Intel i860 CPUs, with internal caches, external cache/buffer, synchronization, and performance monitoring logic as described in section 4. The 256 MMs supply 256 MBytes or 32 M 32-bit words of memory. These memory words are 16-way interleaved in both orthogonal directions as described in section 5.

The SBN is a custom-designed memory backplane. All the 16 MBs are vertically plugged into this backplane. The PBs are connected to the OBN using flat cables. The OBN implements the 16 horizontal and 16 vertical spanning buses as shown in Fig.3b. The 2-way switching between x-access and y-access is built on the backplane. Besides switching, the SBN is designed to buffer high-bandwidth data transfers between the orthogonal memory and the PBs.

In summary, the OMP backend machine is built on 8 PBs, 16 MBs, 1 IB, and SBN backplane. The host board and all the PBs are plugged into the same VME chasis. The purpose of the prototype OMP construction is to prove scalability in performance and yet allow modular growth. In subsequent sections, we identify the research and development issues and present our methods of attacking these issues.

The processing speed of OMP is analyzed below in terms of MIPS (Million Instructions Per Second) rate. This analysis is based on the latency parameters used in the actual design. Let x be the peak MIPS rate of a single *i*860 processor. For an example, $x = 27$ MIPS for a 33MHz i860 [23]. Figure 3c shows a hierarchy of 3 levels of physical memories in the OMP: the *internal cache* inside the *i*860 chip (level 1), the *external cache* on the processor board (level 2), the *local memory* on the processor board (level 3) and the *orthogonal memory* (level 3).

Consider the execution of N instructions on each *i*860 processor in a p-processor OMP system. Let f_p and $f_s = (1 - f_p)$ be the fractions of these instructions used for *processing* and *synchronization* purposes respectively. The memory reference, corresponding to executing $N \times f_p$ instructions, are distributed over the hierarchical memory space seeing from each processor. Let f_{p1}, f_{p2}, f_{p3}, and f_{p4} be the respective fractions of memory references to *internal cache* together with CPU *registers*, *external cache*, *local memory*, and *orthogonal memory*; where $f_{p1} + f_{p2} + f_{p3} + f_{p4} = f_p$. The execution of instructions with memory references to internal cache and registers is determined by the peak MIPS rate x of each *i*860. The execution of those instructions using off-chip resources is limited by memory access times (t_2, t_3, and t_4 as shown in Fig. 3c) and by the *synchronization overhead* t_s. So the *effective MIPS rate*, X, of the OMP is estimated as:

$$X = \frac{p \cdot x}{f_{p1} + (f_{p2} \cdot_2 + f_{p3} \cdot t_3 + f_{p4} \cdot t_4) \cdot x + f_s \cdot t_s \cdot x} \tag{1}$$

We have chosen the following design parameters: $p = 16$ processors, $x = 27$ MIPS per processor, $t_2 = 91$ nsec for external cache/buffer access, $t_3 = 547$ nsec for a 32-byte block data movement from local memory to external cache, $t_s = 6454$ nsec for 16 32-bit interleaved data fetch from orthogonal memory to external cache/buffer, and $t_s = 1243$ nsec for synchronizing all the processors. These parameters are based on a 33-MHz *i*860 processor. Both internal and external caches are block-oriented. The blockwise data movement between different memory hierarchies give rise to high hit-ratios in the internal cache. Suppose we choose

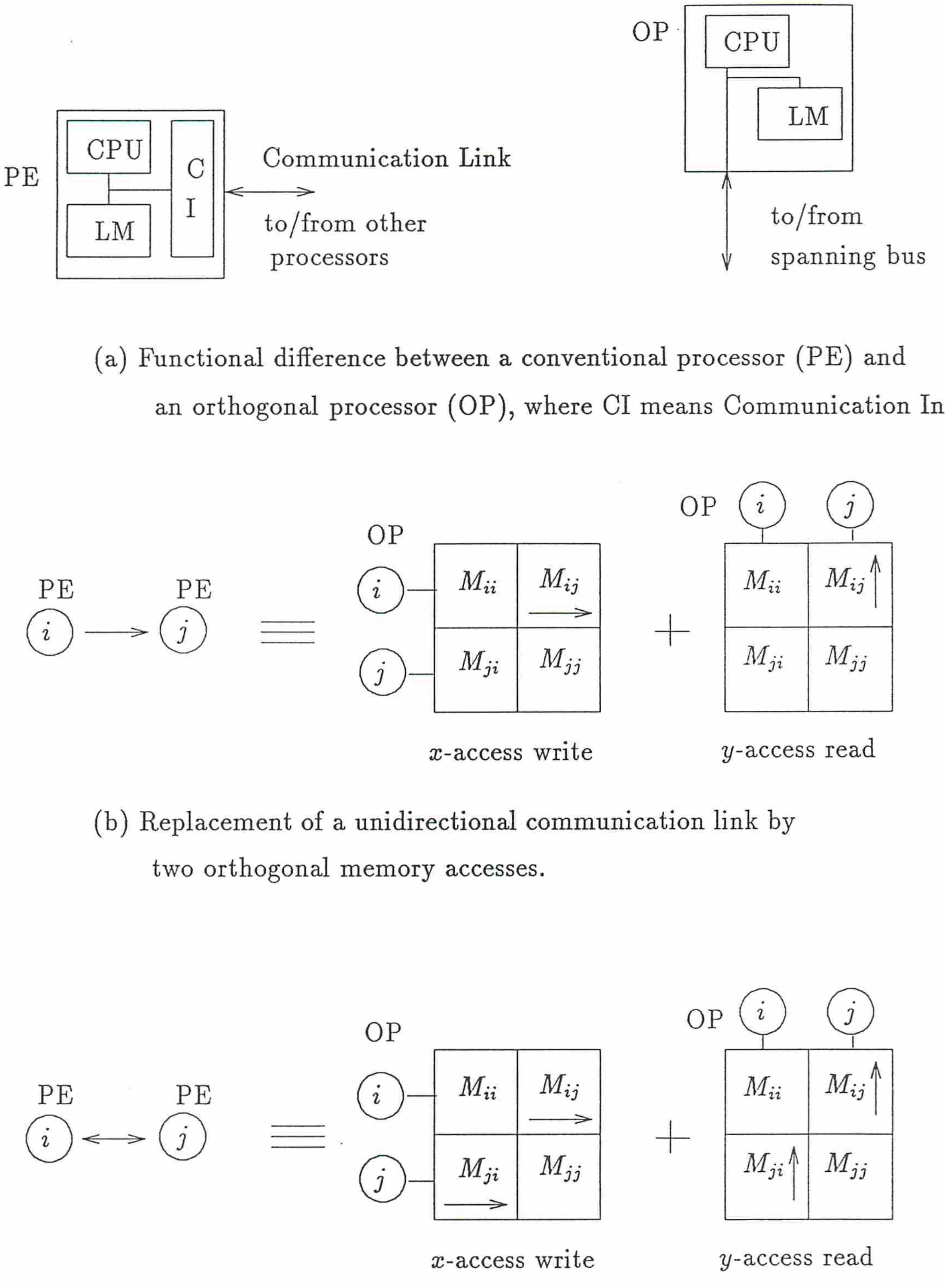
$f_{p1} = 0.95$, $f_{p2} = 0.04059$, $f_{p3} = 0.007425$, $f_{p4} = 0.001485$, and $f_s = 0.0005$ as a test sample. This results in a 301 MIPS rate of the experimental system. This implies that our design could be 70% efficient as compared to the theoretical peak rate of 432 MIPS. In section 6, we will report measured MIPS rates of the OMP based on 16 simulation experiments.

3. Embeddings of Other Architectures

In this section, we reveal superset architectural characteristics of the OMP. A new shared-memory paradigm is introduced for *orthogonal vector communication*. We prove that an n-processor OMP is capable of embedding any other n-processor architecture. The idea of embeddings is illustrated by mapping a 16-processor *hypercube* and a 4 × 4 *mesh* computer into a 16-processor OMP. Each processor can access the memory modules of two neighboring processors using the *shifted memory access*. We also illustrate the embeddings of a 16 × 16 mesh computer and a 64 base *pyramid*. These embeddings make the OMP system dynamically adaptable to many scientific and image processing applications [2, 31, 34].

Consider the differences of an *Orthogonal Processor* (OP) used in our system, as compared with a *Processing Element* (PE) used in a distributed memory multiprocessor system (Fig. 4a). The OP differs from the PE mainly in the area of interprocessor communication. While a PE communicates with other PEs through dedicated communication links, OPs communicate with each other through the orthogonal memory. We embed a distributed memory multiprocessor network into an OMP by allocating the computational tasks of one or more PEs to an OP and by replacing communication links with pairwise-shared memory modules. The embedding process preserves the adjacency and connectivity properties of the multiprocessor network.

When two processors PE$_i$ and PE$_j$ are connected by an unidirectional link (Fig.4b), we can embed them directly into a 2-processor OMP. The unidirectional communication link is established by two orthogonal memory accesses. The first access consists of OP$_i$ writing data to memory module M_{ij} in x-access (indicated by → in Fig.4). The OP$_j$ performs a y-



(a) Functional difference between a conventional processor (PE) and an orthogonal processor (OP), where CI means Communication Interface.

(b) Replacement of a unidirectional communication link by two orthogonal memory accesses.

(c) Replacement of a bidirectional communication link by two orthogonal memory accesses.

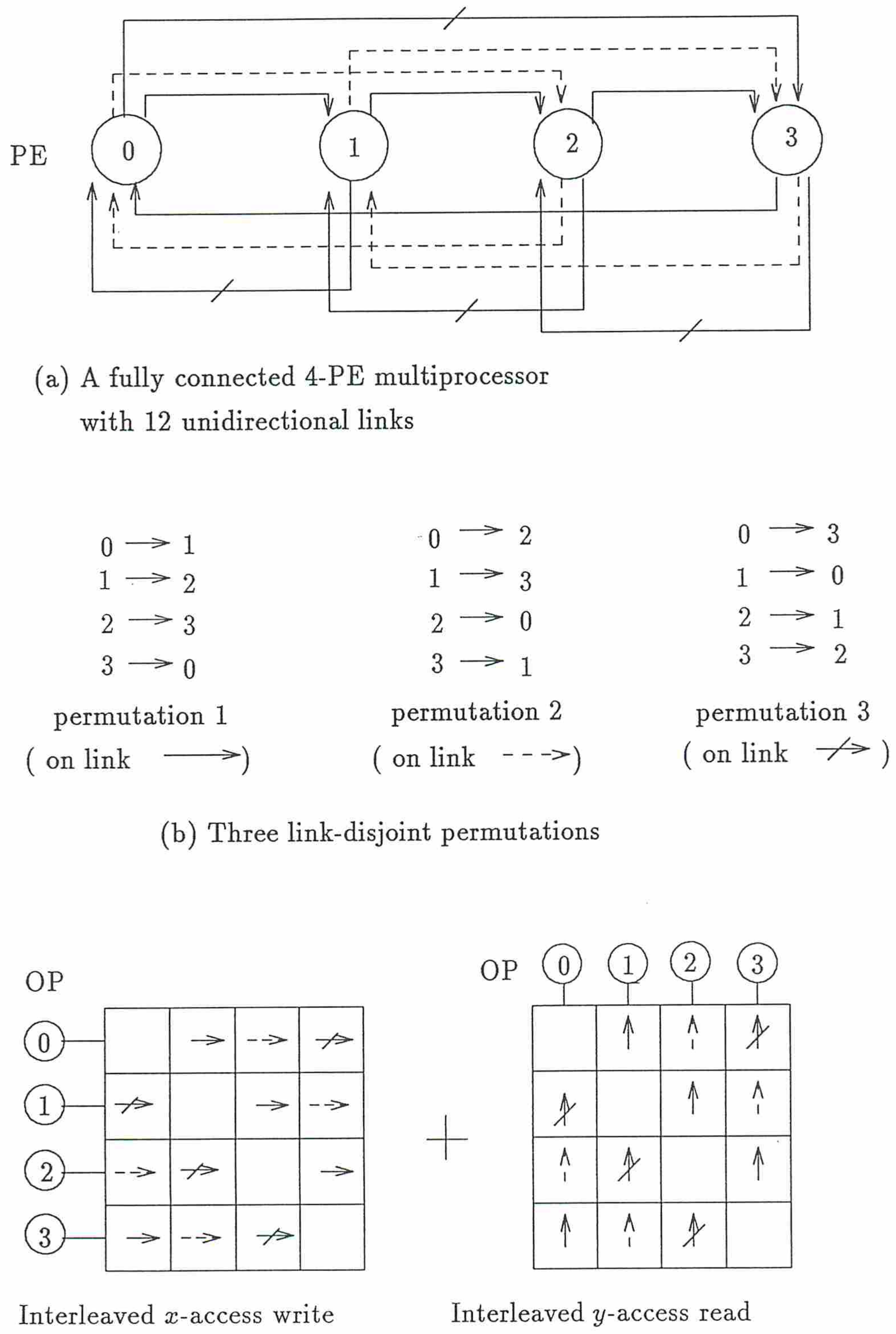
Fig. 4 Replacing communication links of a multiprocessor by orthogonal memory accesses.

access *read* from memory module M_{ij} (indicated by ↑) in the second access. All OPs operate synchronously to access the shared orthogonal memory. This methodology also allows us to replace a bidirectional communication link between two processors by *two-cycle* orthogonal memory accesses as shown in Fig.4c. The OPs perform a *x-access write* followed by a *y-access read* to modules M_{ij} and M_{ji} respectively.

Consider a 4-PE fully connected multiprocessor with 12 unidirectional links as shown in Fig.5a. A PE can communicate with at most 3 other PEs in one time step. We consider the 3 permutations shown in Fig.5b. These permutations can be carried out in one step using dedicated links. In general, an n-PE fully connected multicomputer with $n(n-1)$ unidirectional links can achieve $n(n-1)$ data communications with $(n-1)$ permutations in one step. We show how an n-processor OMP can achieve the same $n(n-1)$ data communications in two memory accesses by an example of 4-processor OMP as shown in Fig.5c. Both x-access and y-access to orthogonal memory are 4-way interleaved in this case.. The *interleaved read/write* allows us to *fetch/store* from/to multiple MMs with the same displacement address in one memory cycle. The two-cycle orthogonal memory access comprises of a *x-access write* followed by a *y-access read*.

The *orthogonal vector communication* paradigm is formally defined below. All possible pairwise communications between OPs can be defined by an ordered set $S = \{(P_i, P_j),\ 1 \leq i, j \leq n\ and\ i \neq j\}$. Besides *permutations*, the set S also contains *one-to-many* and *many-to-one* communications. We define an *orthogonal vector* as $V = \{(P_i, P_j),\ 1 \leq i, j \leq n,\ and\ (P_i, P_j) \subset S\}$. Clearly, V can take $2^{n(n-1)} - 1$ different values. Each V can be implemented by two memory accesses using *interleaved read/write*. This capability makes the orthogonal memory structure even potentially more powerful and cost-effective than a crossbar switching network. In fact, *an OMP can embed directly any distributed-memory multiprocessor system consisting of up to* $n^2 - n$ *unidirectional or* $(n^2 - n)/2$ *bidirectional links*, where n is the number of processors in the system..

Figure 6 shows the embedding of the bidirectional links of a 16-processor hypercube [24] into our orthogonal memories. Each bidirectional link communication is replaced by two

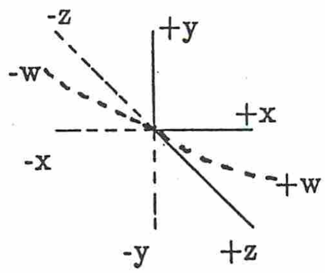


(a) A fully connected 4-PE multiprocessor
with 12 unidirectional links

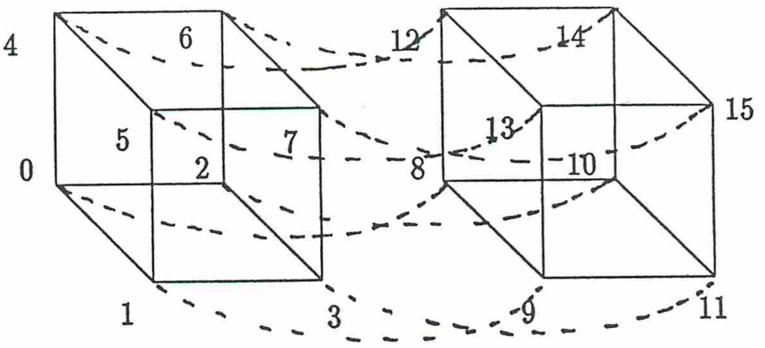
(b) Three link-disjoint permutations

(c) Two-cycle orthogonal communication for
three link-disjoint permutations

Fig. 5 Embedding a 4-PE fully connected multiprocessor
into a 4-OP orthogonal multiprocessor.



4 dimensional links of a node



A 16 node hypercube

↱ = A *x*-access *write* followed by a *y*-access *read*

OP	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		+z ↱	+x ↱		+y ↱				+w ↱							
1	-z ↱			+x ↱		+y ↱				+w ↱						
2	-x ↱			+z ↱			+y ↱				+w ↱					
3		-x ↱	-z ↱					+y ↱				+w ↱				
4	-y ↱					+z ↱	+x ↱						+w ↱			
5		-y ↱			-z ↱			+x ↱						+w ↱		
6			-y ↱		-x ↱			+z ↱							+w ↱	
7				-y ↱		-x ↱	-z ↱									+w ↱
8	-w ↱									+z ↱	+x ↱		+y ↱			
9		-w ↱							-z ↱			+x ↱		+y ↱		
10			-w ↱						-x ↱			+z ↱			+y ↱	
11				-w ↱						-x ↱	-z ↱					+y ↱
12					-w ↱				-y ↱					+z ↱	+x ↱	
13						-w ↱				-y ↱			-z ↱			+x ↱
14							-w ↱				-y ↱		-x ↱			+z ↱
15								-w ↱				-y ↱		-x ↱	-z ↱	

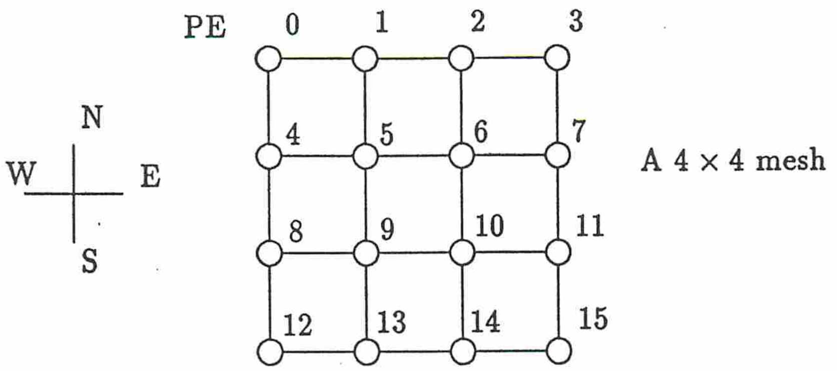
Fig. 6 Embedding a 16-PE hypercube with 32 bidirectional links into the 256 orthogonal memory modules of a 16-processor OMP.

memory accesses(a x-access *write* $\rightarrow$ followed by a y-access *read* $\uparrow$) through two symmetric shared memory modules. For example, module M_{19} is used for P_1 to communicate with P_9 in the $+w$ dimension and M_{91} for P_9 to P_1 in the $-w$ dimension. This embedding scheme provides the flexibility of implementing hypercube communications in a single dimension $(+x/+y/+z/\ldots/-w)$ or in any groups of multiple dimensions $((+x,+y)/(+y,+z,-w)/\ldots$ etc) simultaneously.

Figure 7 shows the embedding of a 4 × 4 mesh [26] of 16 PEs into the 256 MMs of a 16-processor OMP. The 24 bidirectional links are replaced by shared memory communication via 48 shared memory modules. For example, bidirectional communication between PE_{10} and PE_{11} is replaced by accessing the $M_{10,11}$ for *eastbound* (E) and the $M_{11,10}$ for *westbound* (W) data movement. The embedding also provides flexibility to implement one or more of the E, W, N, or S data movements among 16 processors simultaneously by accessing the respective MMs.

The bus switching logic is built on the memory backplane. An OP can access not only memory modules on its own spanning bus, but also on either of its two neighboring buses. Besides the *normal* y-access, the OPs can perform *right-shift y-access* or *left-shift y-access* synchronously. Similar *shifted access* (up and down) are also allowed in x-access mode. The shift operations take place in a wrap-around fashion. Figure 8b and 8c show examples of *right-shift y-access* and *down-shift x-access* respectively.

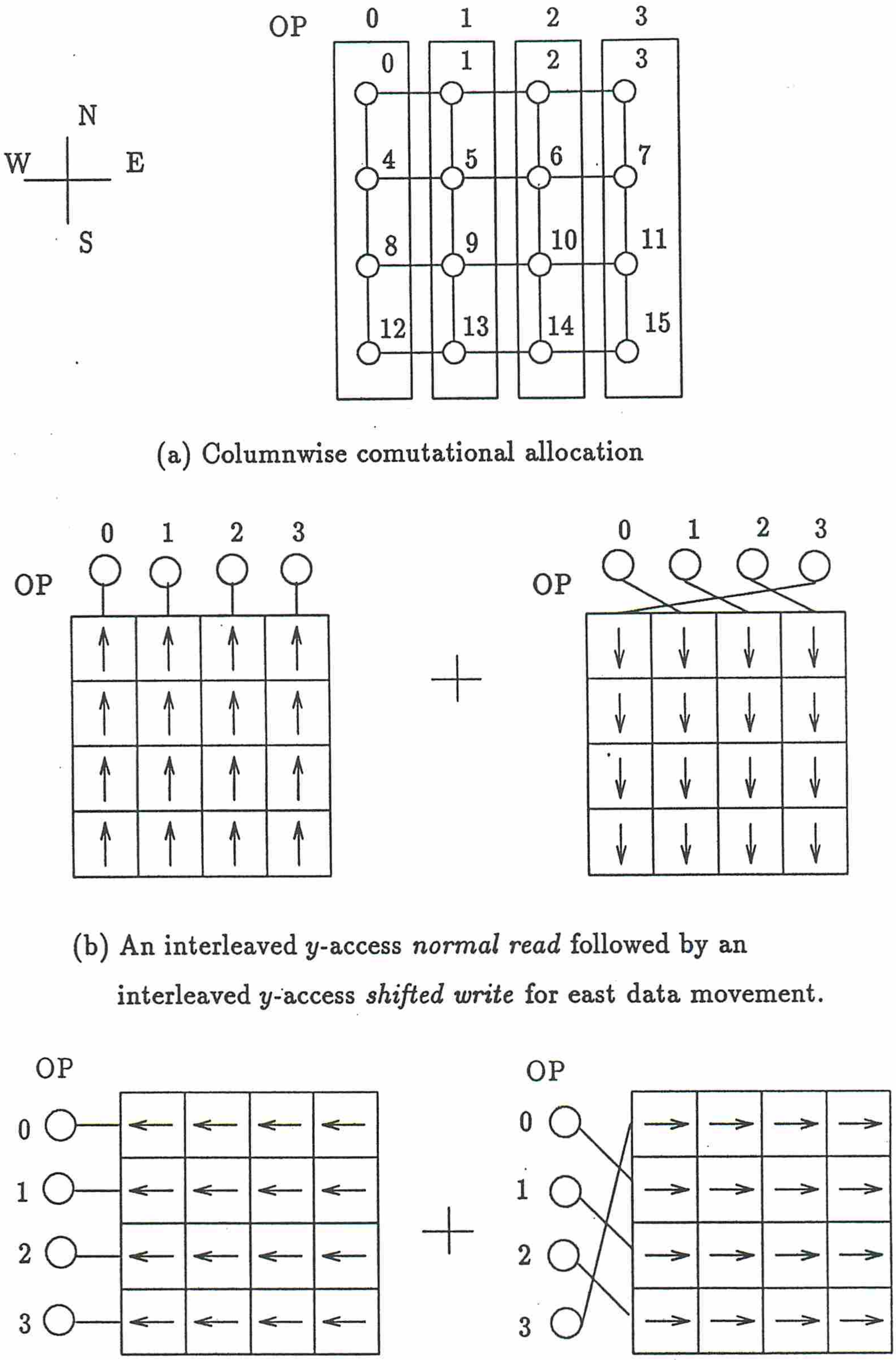
This bus switching scheme makes the OMP architecture more powerful. Figure 8 shows the embedding of a 4 × 4 mesh of 16 PEs into a 4-processor OMP. The computation tasks of 4 PEs in each column are allocated to a single OP. The E data movement of the mesh can be carried out by two orthogonal memory accesses. In the first cycle, all processors perform interleaved y-access *normal read* operation (indicated by $\uparrow$ in Fig.8b). In the second cycle, all processors perform *interleaved write* operation (indicated by $\downarrow$) using y-access right-shifted buses. Similar operations are needed for W data movements. Figure 8c shows the two cycle operation for southbound (S) data movement.



→↑ = A x-access *write* followed by a y-access *read*

OP	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		E→↑			S→↑											
1	W→↑		E→↑			S→↑										
2		W→↑		E→↑			S→↑									
3			W→↑					S→↑								
4	N→↑					E→↑			S→↑							
5		N→↑			W→↑		E→↑			S→↑						
6			N→↑			W→↑		E→↑			S→↑					
7				N→↑			W→↑					S→↑				
8					N→↑					E→↑			S→↑			
9						N→↑			W→↑		E→↑			S→↑		
10							N→↑			W→↑		E→↑			S→↑	
11								N→↑			W→↑					S→↑
12									N→↑					E→↑		
13										N→↑			W→↑		E→↑	
14											N→↑			W→↑		E→↑
15												N→↑			W→↑	

Fig. 7 Embedding a 16-PE mesh into the 256 orthogonal memory modules of a 16-processor OMP.



(a) Columnwise comutational allocation

(b) An interleaved *y*-access *normal read* followed by an interleaved *y*-access *shifted write* for east data movement.

(c) An interleaved *x*-access *normal read* followed by an interleaved *x*-access *shifted write* for south data movement

Fig. 8 Embedding of a 4 × 4 mesh (with wrap around) into the orthogonal memories of a 4-processor OMP.

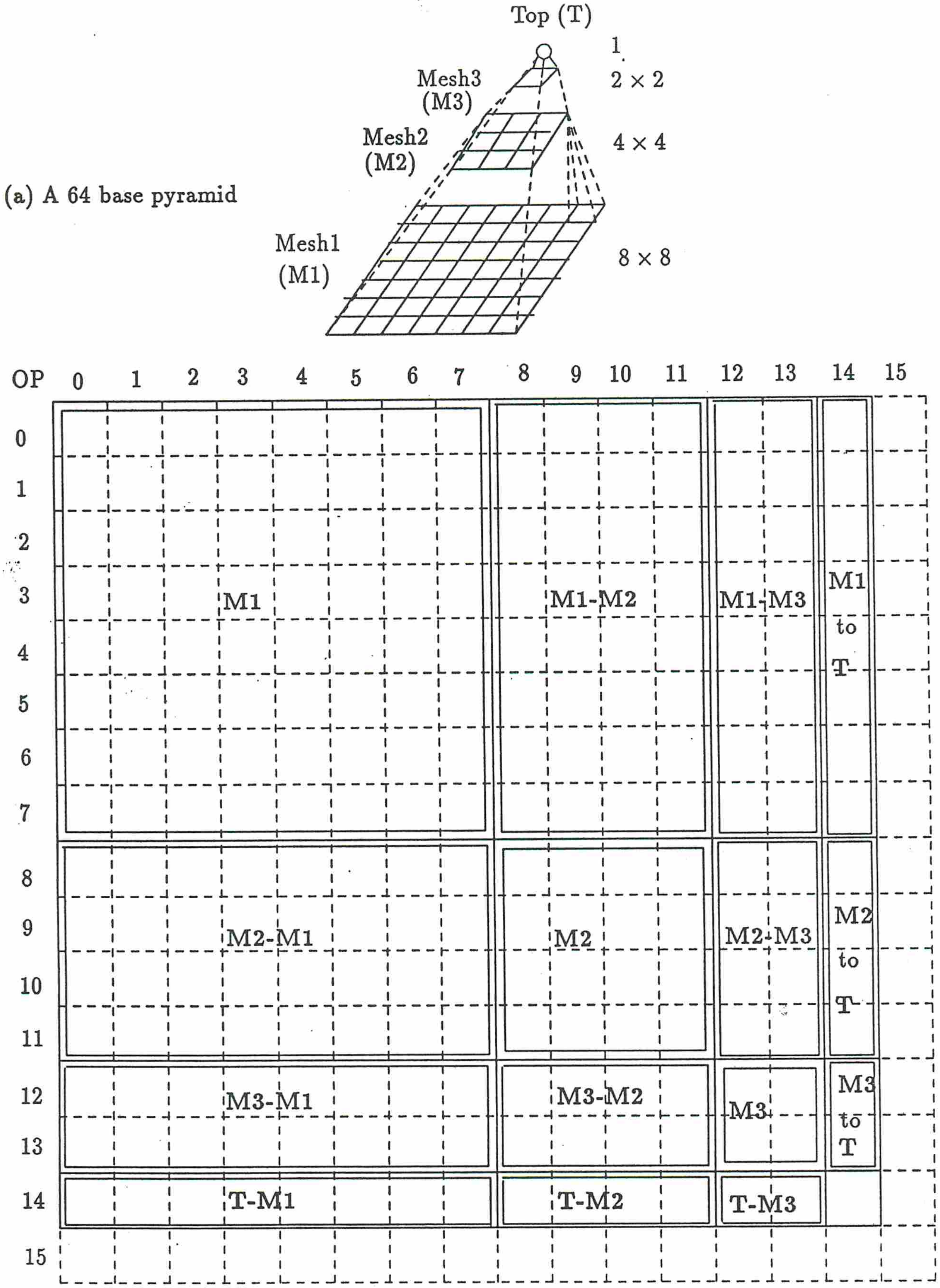
Finally, we illustrate in Fig.9 how to embed a hierarchical *pyramid* architecture into an OMP. The pyramid consists of 3 decreasing meshes M1, M2, M3 of sizes 8×8, 4×4, and 2×2 respectively. These meshes can be embedded into the 16 OPs by assigning $8, 4, 2$ OPs to meshes M1,M2, and M3 respectively. The Top (T) processor of the pyramid is assigned to a single OP. The intermesh communications are carried out by blocks of diagonal MMs (blocks marked by M1, M2, and M3 in Fig.9). The intramesh communications, both upward and downward, are carried out by off-diagonal blocks of MMs.

For example, the upward communication between meshes M1 and M2 is carried out by the 8×4 MMs marked by block M1-M2. Similarly, downward communications between meshes M3 and M2 are carried out by block M3-M2. This embedding allows concurrent *intermesh, upward*, and *downward* communications. It also supports nonadjacent mesh communications (for example M1 to M3, T to M2, T to M1 etc), which are difficult to implement in a normal pyramid computer [1, 37].

4. Processor Board with Vector Extensions

There are two i860 processors mounted on each *processor board* (PB). Each processor communicates with other PBs through a shared VME interface and accesses the orthogonal memory through a special memory interface on the PB. As shown in Fig.10, each processor uses a 2-way, set associative, *external cache*, through which it accesses the $512K \times 64$ *local memory* and the off-board *orthogonal memory.* Each processor can access a row of 16 MMs using *x-access* or a column of 16 MMs using *y-access.*

The major components on the PB include two *processor modules*, a *shared VME bus interface*, an *orthogonal memory interface*, *connectors for cables* connected to the orthogonal memory backplane and two *serial ports.* Each processor module consists of one i860 processor, its Boot-EPROM, local memory, external cache and a serial port interface to an RS-232 port. The i860 was chosen because of its unique features, including a 64-bit *external data bus*, an *internal* 128-bit *bus*, a high speed *floating point unit* capable of executing two



(a) A 64 base pyramid

(b) Block-oriented allocation of orthogonal memory modules

Fig. 9 Embedding of a 64-base pyramid into the orthogonal memory modules of a 16-processor OMP.

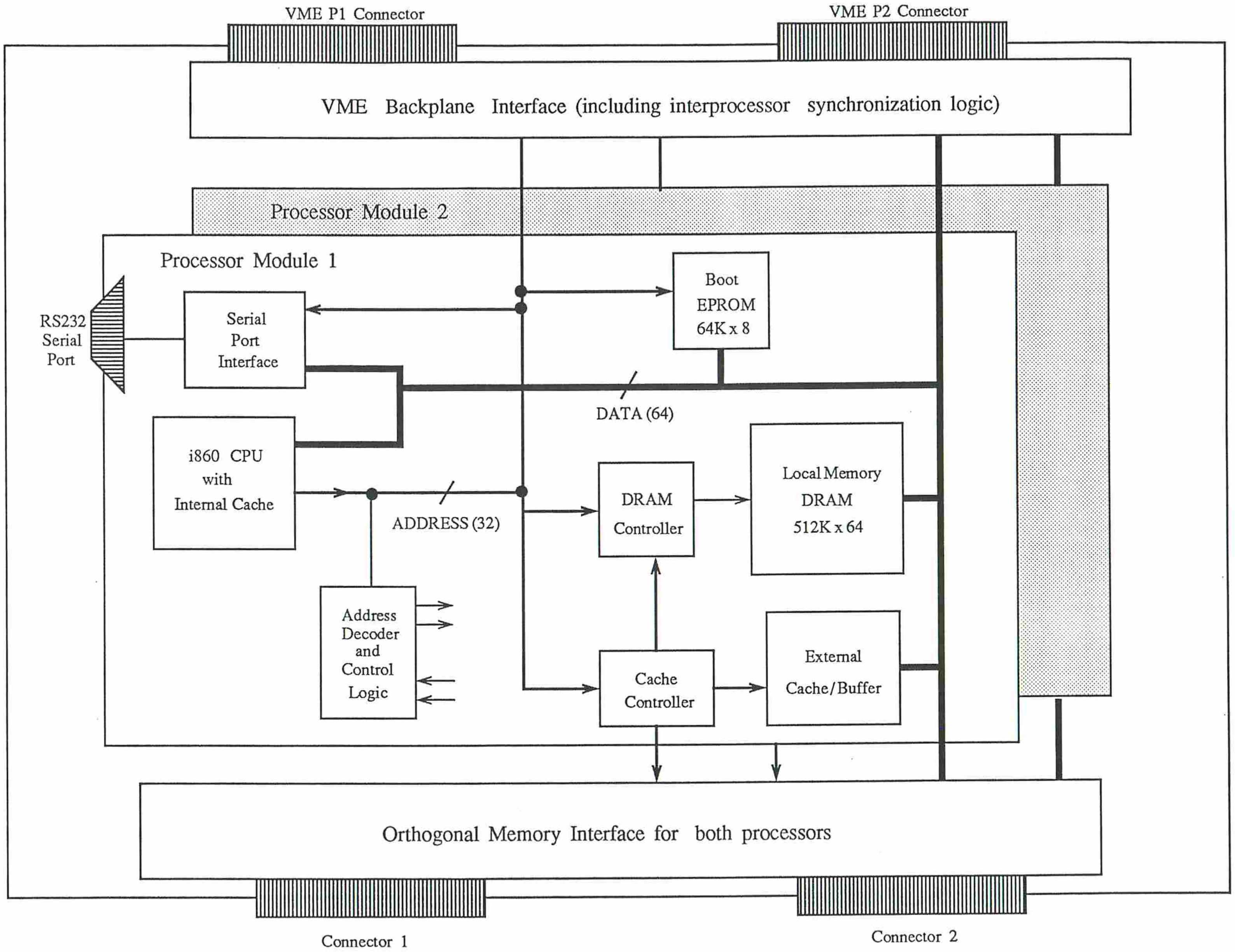


Fig. 10 Major components in dual processor modules on the processor board

floating point operations per clock and the capability to perform *vector operations* with the aid of a vector library of subroutines. Besides this, it has a *graphics unit* operating at 16 million 16-bit pixels/sec, which supports image display operations in the OMP. At its peak rate of operation, the i860 can sustain two floating point operations and one integer RISC operation per clock.[23]

Each processor is designed to boot up from a $64k \times 8$ EPROM. The EPROM holds the *self-test code*, the *monitor program*, a *loader* for downloading code from the host to the PB, and some *initialization procedure* for the PB. The 4 Mbytes of local memory for each processor is organized as $512K \times 64$ with 80 nsec DRAMs. This implies that the processor accesses local memory at 33 MHz with 4 wait states. The processor accesses one 64-bit word per read/write cycle giving a bus bandwidth of 44 Mbytes/sec for local memory. The program and run-time scalar data for the i860 resides in the local memory. This includes a stripped-down version of the MACH/OS to support orthogonal multiprocessing, the user program and data sets. All this is downloaded from the host over the VME bus.

The external cache on the PB acts as a fast interface between the i860 processor and its local memory as well as to the orthogonal memory. An 8K bytes, 2-way, set-associative internal data cache allows one clock access of up to 128 bits of data. A *miss* in the internal cache goes to the external cache. This is a 8-words-per-block and 2-way, set-associative cache, with a line size of 512 bits. This is implemented with fast static RAMs with 25 ns access time, permitting 64-bit zero-wait-state accesses. All external caches are private to the processors attached.

A *miss* in the external cache is directed to the orthogonal memory interface. This initiates a request to access the interleaved memory over the orthogonal memory bus. Both internal and external caches are flushed when the processors switch between x-access and y-access. Both caches employ a *write-back* policy. Estimating the memory access times, we have one clock per 128 bits for the internal cache, 3 clocks per 64 bits for the external cache and 16 clocks per 32 bits for the orthogonal memory.

Let B_1 be the bandwidth of the internal processor bus for internal data cache accesses, B_2 be the bandwidth of the PB data bus for external cache accesses, B_3 be the bandwidth of an orthogonal bus for orthogonal memory accesses and B_4 be the bandwidth of the PB data bus for local memory accesses. Also, let h_1 be the hit ratio for internal data cache accesses, h_2 be the hit ratio for external cache accesses and f the probability of accessing an address in orthogonal memory during a processor's memory cycle. Then, $(1 - f)$ represents the probability for the processor to access an address in local memory during a memory cycle. In terms of these parameters, the effective bandwidth for one processor to access the memory system consisting of the on-board local memory and the off-board orthogonal memory is,

$$B^1_{eff} = \frac{1}{\frac{h_1}{B_1} + \frac{(1-h_1)h_2}{B_2} + (1-h_1)(1-h_2)[\frac{f}{B_3} + \frac{(1-f)}{B_4}]} \tag{2}$$

For the entire system consisting of n processors, the effective bandwidth for memory accesses should be, $B^n_{eff} \leq n \times B^1_{eff}$. Table 2 shows the potential bandwidth for the processor to access the orthogonal memory using Eq. 2.

Table 2: Potential Memory Bandwidth Per Processor as a Function of Cache Hit Ratios, $(f = 1,\ 0 \leq x \leq 1)$

Hit Rate		*Effective Bandwidth*
Internal Cache, h_1	*External Cache,* h_2	*Per Processor* B^1_{eff} *Mbytes/sec*
1.0	x	528
0.9	0.9	261
0.9	0.7	173
0.7	0.9	130
0.5	0.5	32
0.1	0.1	11
0	x	9

For interprocessor synchronization at a memory access level the *lock* bit of the i860 is used to implement mutual exclusion. In addition, all processors have a common synchronization point for orthogonal memory access at the time of switching between x-access and y-access modes. Interprocessor communication over the VME bus serves as higher level, interrupt-based, synchronization among processes running on different processors.

To support vector processing using the orthogonal memory, it is necessary to define vector operations to be executed by the RISC i860 CPU. The vector library supplied by Intel consists of some subroutines designed to optimize the pipelining operations. However, the Intel-supplied vector routines are not sufficient for our purpose. Therefore, we decided to develop additional vector operations in i860 assembly code, specially tailored for OMP applications. Some of these vector operations are shown in Table 3.

Table 3: Some Vector Operators Needed For The OMP

Operator	*Notation*	*Result*
1. Vector Add	$C = A + B$	vector
2. Vector Subtract	$C = A - B$	vector
3. Vector Multiply1	$c = A \ . \ B$	scalar
4. Vector Multiply2	$C(m \times n) = A(m \times 1) \ X \ B(1 \times n)$	array of vectors
5. Vector Sum	Sigma(A)	scalar
6. Vector Max	MAX(A)	scalar
7. Vector Min	MIN(A)	scalar
8. Vector Load	LOAD(A)	vector
9. Vector Write	WRITE(A)	vector

As an example, we show how *Vector Multiply2*, the cross-product of two vectors, is implemented in a pseudo i860 code as listed in the appendix. In this operation, A is an m x 1 column vector and B is a 1 x n row vector. The multiplication of A and B produces an m x n matrix C. Due to the three stages in the multiply pipeline for single precision multiplication, the result of the current multiply instruction will go to the destination register (dest0, dest1, etc.) specified by the third subsequent multiply instruction. Also, due to the pipeline optimization, arrays A, B and C may be accessed beyond their logical limits. The i860 performs each pair of instructions prefixed by a *d.* in one clock when pipelined.

The PB is designed to run at 33 MHz, matching a 264 Mbytes/sec access rate for the internal instruction cache, B_1 = 528 Mbytes/sec for the internal data cache, B_2 = 88

Mbytes/sec for the external cache and $B_4 = 44$ Mbytes/sec for the local memory. For 16 clocks per 32-bit data access, the orthogonal memory has a worst case bandwidth of $B3 = 9.1$ Mbytes/sec. The current implementation has two i860 processors per PB due to board area and connector restrictions. There is a future possibility of expanding this to four processors per PB. The PB is being designed on the *Viewlogic* schematic capture system, [38], and timing analysis is being performed using the *Verilog* simulator [13]. We are presently using STAR 860, an i860 software development system, [22], in extending the vector routines, developing software support for interprocessor synchronization and interrupts, resource management, fault diagnosis and performance monitoring using the serial port attached to each processor.

5. Orthogonal Memory and Spanning Buses

The spanning buses between the i860 processors and the orthogonal memory are built on the memory backplane. The memory access controller and switching logic for the spanning buses are mounted on the same backplane. The memory controller controls two modes of orthogonal memory access, namely the *x-access* (or *row access*) and *y-access* (or *column access*). The bus switching logic consists of multiplexers/demultiplexers which switch bus B_i of processor P_i to its corresponding x-bus, B_i^x, or y-bus, B_i^y. Processor P_i accesses MMs M_i^x in row i through B_i^x using x-access, and MMs M_i^y in column i through B_i^y using y-access.

In addition, two *shifted memory access* modes are implemented, in which each processor accesses memory modules on one of its two neighboring buses. *i.e.* processor P_i accesses modules M_{i-1}^y through bus B_{i-1}^y in a *left-shift y-access* or modules M_{i+1}^y through bus B_{i+1}^y in a *right-shift y-access*. Similarly, processor P_j accesses modules M_{j-1}^x through bus B_{j-1}^x in an *up-shift x-access* or modules M_{j+1}^x through B_{j+1}^x in a *down-shift x-access*. These shifted accesses operate in a wrap-around fashion as shown in Fig.8b and 8c. This mechanism is very useful to embed several architectures into the OMP as shown in Section 3, as well as to implement image processing algorithms based on window operations efficiently.

To simplify the routing of the buses on the backplane, every 4 adjacent x-buses, B_i^x to B_{i+3}^x, $(i = 0, 4, 8, 12)$ are time multiplexed into one physical bus, called *Horizontal bus* (*H-bus*). Similarly every 4 adjacent y-buses, B_i^y to B_{i+3}^y, $(i = 0, 4, 8, 12)$ are multiplexed into a *Vertical bus* (*V-bus*). Thus for our 16-processor OMP, there are 4 H-buses and 4 V-buses on the backplane. Figure 11 shows the interconnections among the memory boards through the H-buses and V-buses. This multiplexing scheme has the advantage of reducing the number of backplane connectors on the memory board and hence the size of the *memory board* (MB).

The *orthogonal memory* of our prototype OMP system consists of 16 MBs. One MB houses 16 MMs (a 4×4 subarray of the 16×16 array shown in Fig.3b). Each MB is connected to a H-bus and a V-bus on the backplane. Memory accesses take place through the H-bus in x-access mode and through the V-bus in y-access mode. In order to make the layout of all the MBs identical, four V-bus connectors are provided on every board. This also permits us to plug in an MB into any slot on the backplane. Depending on which slot the board is plugged into, it will be connected to one of the four H-buses and one of the four V-buses. For purposes of identification, an MB may be specified as $\text{MB}_{H,V}$ where the H and V indicate which H-bus and V-bus the board is connected to. Figure 12a shows the layout of a memory board, $\text{MB}_{1,3}$.

Each of the 16 MMs on a board has a capacity of 1 Mbyte, organized as 256K $\times$32 bits. The total capacity of the orthogonal memory is thus 256 Mbytes. Even though the word size of the memory does not match with the 64 bit width of the processor bus, this word size is sufficient to support single-precision arithmetic and image processing applications. Double-precision data are stored in two consecutive words within an MM. Since each module M_{ij} receives requests from the H-bus or the V-bus, the address and control signals from the two buses are multiplexed. The data lines, due to their bidirectional nature, go through a multiplexing/demultiplexing stage. The logical blocks within an MM are shown in Fig.12b.

Various parallel memory organization schemes exist for accessing vector data at high speed [9, 25, 17, 28]. We implement high speed vector data accesses from/to the orthogonal memory array by using a 2-dimensional, 16-way interleaving scheme. Vector operands are

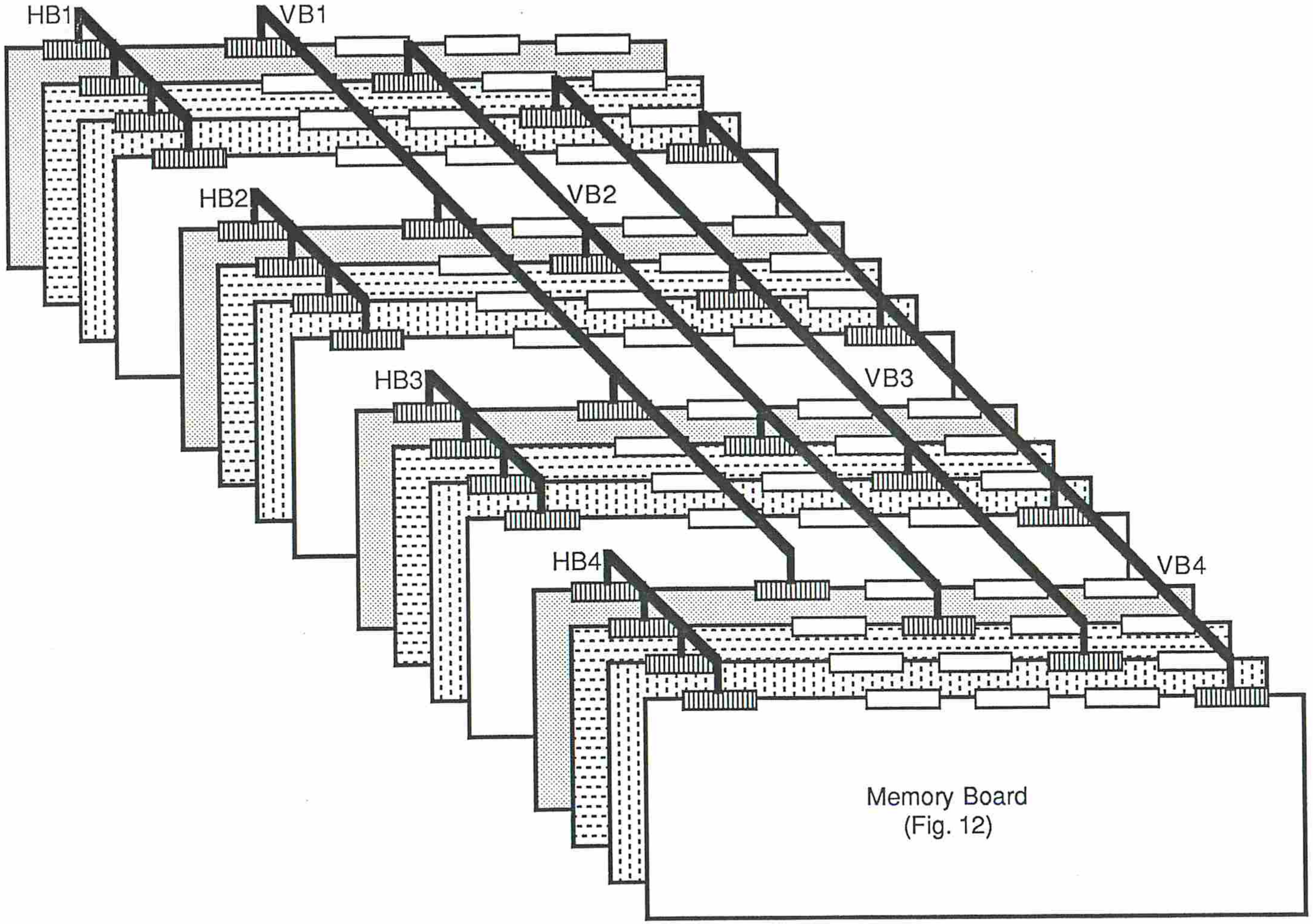
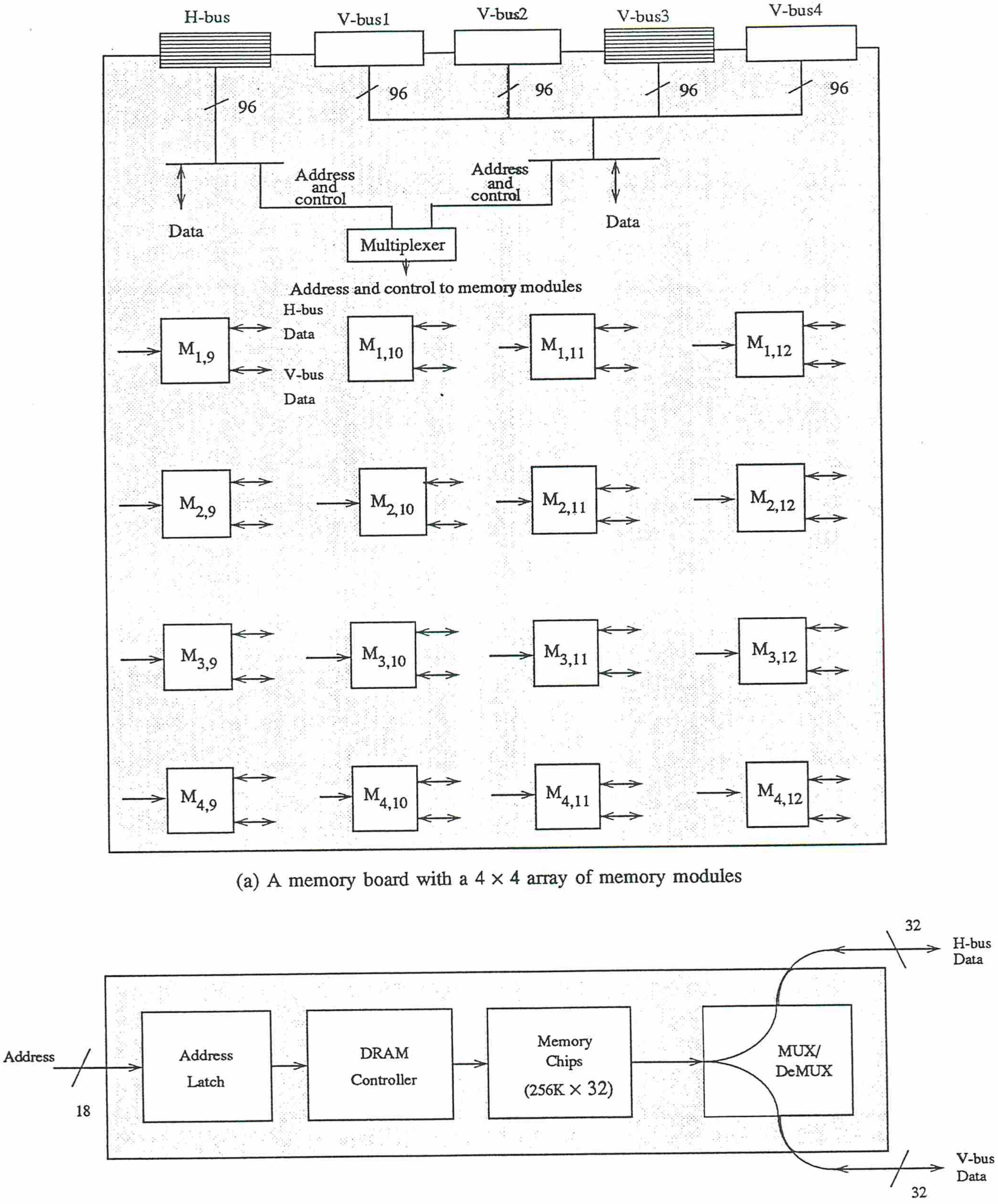


Fig. 11 Backplane bus interconnections among the 16 memory boards, where the HB_i are horizontal buses and VB_i are vertical buses spanning into the x- access and y-access directions respectively.



(a) A memory board with a 4 × 4 array of memory modules

(b) A memory module

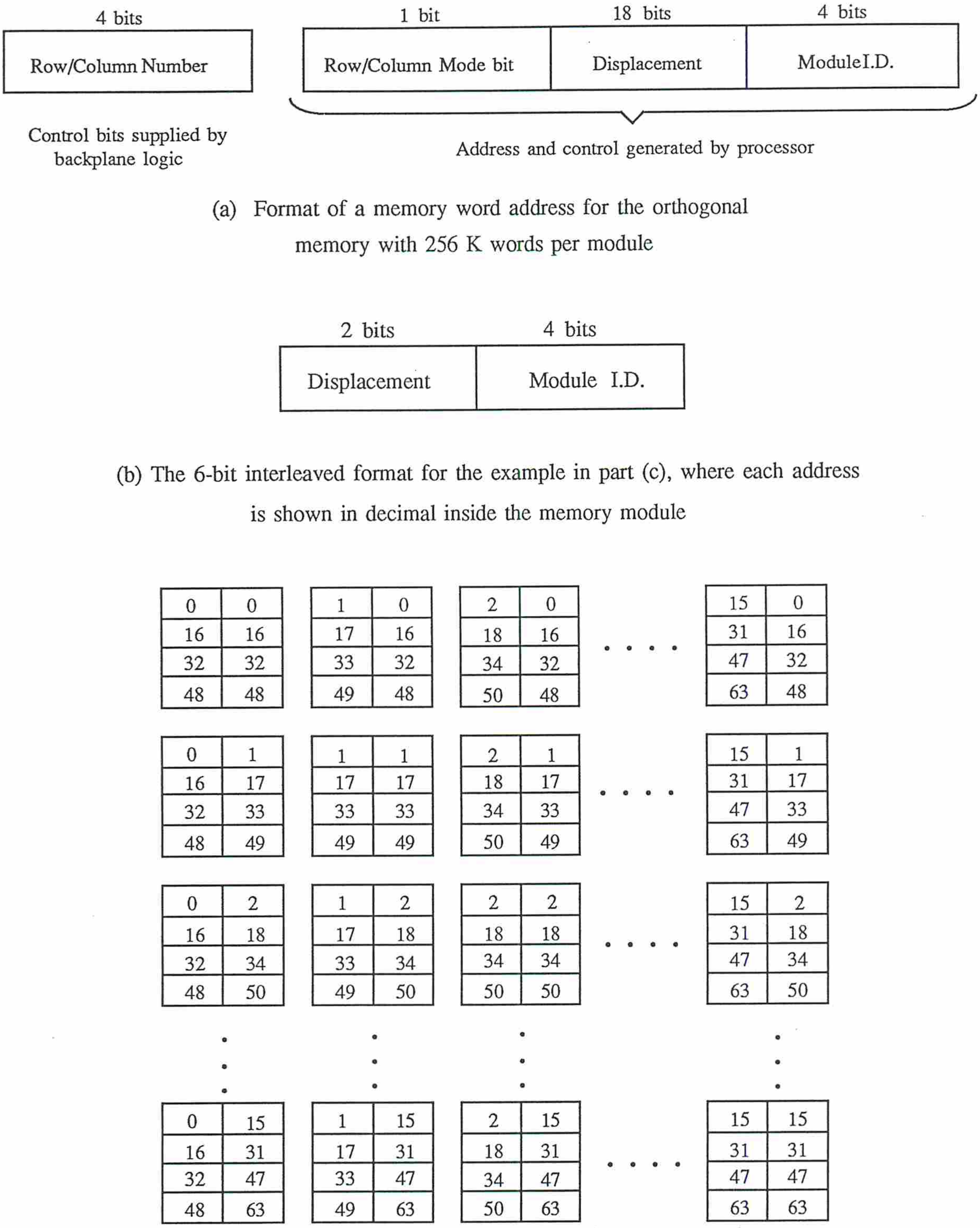
Fig. 12. Functional organization of a memory board

interleaved across the 16 MMs in each of the two orthogonal directions. Each processor accesses an entire row or entire column (16 elements) of vectors in a pipelined fashion. In either of the two orthogonal accesses, processor P_i has access to the modules M_i^x or M_i^y. These modules are identified sequentially from 0 through 15. Furthermore, the addressing scheme must be identical along either the x or the y directions. This means that each memory word in module $M_{ij}(i \neq j)$ should have two addresses: an address A_x when accessed in the x-access mode by processor P_i and an address A_y when accessed in the y-access mode by processor P_j. A_x and A_y have the same displacement, but differ in the module identity. We provide a hardware implementation scheme to manipluate these two addresses automatically.

The address format of a memory location is shown in Fig.13a. The complete address of a memory word consists of the memory module identity (I.D.), the displacement of that word within the module, a bit to indicate row/column access mode and the row/column number. The row/column number is generated by the memory controller to support multiplexing/demultiplexing on H-bus or V-bus. Sixteen-way interleaving is achieved by using the 4 least significant bits of the address for module identification. Figures 13b and 13c show the interleaved addressing in x-access mode (on the left hand side) and y-access mode (on the right hand side) for an orthogonal memory array having four words per memory module.

The orthogonal memory and spanning buses are designed to allow each i860 processor to fetch a vector of sixteen 32-bit elements from orthogonal memory to external cache within 213 clocks or 6454 nsec. These timings are based on a 33 MHz i860 processor. Since a processor can access an 8-byte word from external cache with one wait-state as shown in Section 4, the effective access time of a 64-bit word from othogonal memory is $[213 + (7 \times 3)]/8 = 29.25$ clocks or 886.4 nsec. Thus the orthogonal memory bandwidth per processor is estimated as 9.03 Mbytes/sec.

Besides *x-access* and *y-access* of columns and rows, we show four types of data manipulation using the orthogonal memory. These correspond to *data broadcast*, *remote access*, *diagonal access* and *exchange of rows and columns* as defined below:



(a) Format of a memory word address for the orthogonal memory with 256 K words per module

(b) The 6-bit interleaved format for the example in part (c), where each address is shown in decimal inside the memory module

(c) The 16-way interleaved memory addresses where each MM is illustrated with 4 words and each word is x-accessed by the left address and y-accessed by the right address

Fig. 13 The 2-Dimensional,16-way memory interleaving in the orthogonal memory organization

Table 4: Steps For Data Broadcast, Remote Row Access and Diagonal Access

Data broadcast in two memory cycles	Remote row access in three memory cycles	Diagonal access in three memory cycles
x-*access* mode;	y-*access* mode;	y-*access* mode;
For processor P_i do	**Forall** processors $P_k(k = 0, 1, \ldots, p-1)$	**Forall** processors $P_k(k = 0, 1, \ldots, p-1)$
read w; /* from local memory */	**doparallel**	**doparallel**
Interleaved write w to all M_i^x;	Vector read A_k^y from M_k^y;	Vector read A_k^y from M_k^y;
y-*access* mode;	For $i = 1$ to p do	For $i = 1$ to p do
Forall processors $P_k(k = 0, 1, \ldots p-1)$	$b_{i,k} = a_{(i+c)\, mod\, p,k}$;	$b_{i,k} = a_{(i+k)\, mod\, p,k}$;
doparallel	Vector write B_k into M_k^y;	Vector write B_k into M_k^y;
Read w from M_{ik};	x-*access* mode;	x-*access* mode;
Interleaved write w to all M_k^y;	Vector read A_k^x from M_k^x;	Vector read A_k^x from M_k^x;
Endforall;	**Endforall**;	**Endforall**;

(A) *Data Broadcast*: A data word w contained in local memory of processor P_i is broadcast to all MMs as shown in the left column of Table 4.

(B) *Remote row/column access*: Processor P_i accesses M_j^x or M_j^y where $j = (i + c)\, mod\; p$ as described in the middle column of Table 4.

(C) *Diagonal access*: Processor P_k accesses a diagonal vector from memory modules $M_{i,(k+i)}$ $(i = 0, 1, \ldots, p - 1)$, as shown in the right column of Table 4.

(D) *Exchange between two rows or two columns*: Scalar data in local memory may be exchanged between two processors P_i and P_j over the VME bus. However, two rows or two columns are exchanged more efficiently using orthogonal memory. An exchange operation using orthogonal memory is similar to a remote access described in (B) above, except that processor P_i accesses row j and P_j accesses row i.

6. OMP Simulator and Performance Results

We have developed an OMP simulator using CSIM in a Unix environment [32, 33]. CSIM has been implemented as a set of extensions to the C programming language. First we introduce the reader to the salient features of CSIM. Then we discuss the design of the OMP simulator and describe some CSIM macros specially developed for performance simulation of OMP. Performance results on sorting and matrix algorithms are presented with interpretation.

In a process-oriented simulator, the computer system is modeled as a set of interacting *processes* (right hand of Fig.14). These processes are executed in parallel with each other. Memories, processors, buses and input-output devices are defined as system *facilities* (left hand of Fig.14). The dynamic interaction between processes and facilities is coordinated by the occurrence of *events*. We define an *event* as an instantaneously occurring change of state of some process or facility.

In CSIM, a process is implemented as a reentrant program with its own private data area. Once initiated, a process can be in one of three states: *active* (currently being

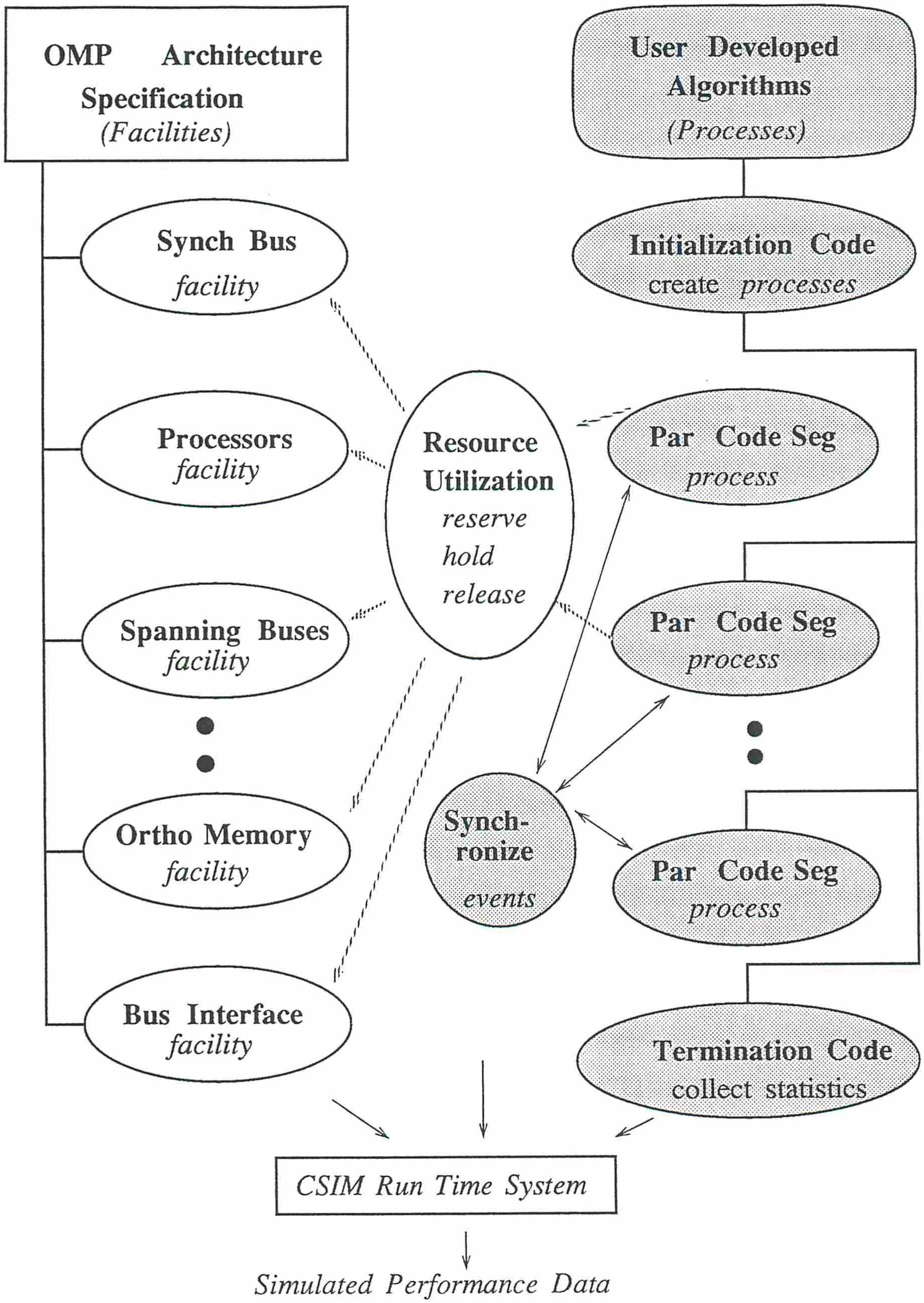


Figure 14: Functional block diagram of the CSIM-based OMP simulator

processed by the simulator), *holding* (waiting for an interval of simulated time to pass), or *waiting* (in a queue for an event to occur). Facilities are treated as passive objects that can assume one of two states, *busy* or *free*. Events also have two states, *occurred* and *not-occurred*. CSIM provides the user with special descriptors for defining processes, facilities and events. An underlying simulation run-time system manages the scheduling and synchronization of processes, management of facilities, and advancement of the global simulation time. Simulated time passes only when processes are in the *hold* state. Upon completion of a simulation run, statistics gathered by the run-time system are reported.

The design philosophy used in our simulator can be characterized as *algorithm-driven simulation.* This design style is adapted from the ASPOL-based simulator for the ETA-10 supercomputer [12]. The OMP simulator introduces the user to the C language extensions provided in Ortho-C as well as to data structures matching the architecture of OMP. With this simulator, the user can experiment with vector data allocation to the shared orthogonal memory, and synchronization on semaphores, etc. Thus the user writes OMP programs with an augmented C language. These programs, after compilation and execution using the CSIM run-time simulation system, generate timing estimates for OMP machines of various sizes.

Within the simulator the main components of OMP have been declared as CSIM *facilities.* These include the processors (i860s), the orthogonal buses, and the orthogonal memory modules, etc. Parallel segments of the algorithm executing on the OMP processors are written as CSIM *processes.* During execution these processes reserve and release the facilities and synchronize their actions using event variables(Fig.14). Whenever a facility is used, simulated time is advanced to reflect its utilization. This corresponds to the delays associated with the actual hardware. Critical OMP hardware component delays have been assumed based on the actual hardware design experience. Examples include orthogonal memory module access time, synchronization bus access time, local memory access time, and RISC instruction execution time as revealed in previous sections. These numbers are progressively refined, as the OMP hardware design makes progress.

Two major extensions to C are introduced. One set consists of *verbs* that will actually be supported by the Ortho-C compiler. The other set consists of *verbs* that are used only for simulation. All these extensions have been encapsulated as *macros* of CSIM statements. Consider the following *macro* example:

```
#define     TSETCNT(x,n)       if (cntr[x]- >init==0) {
                               cntr[x]- > c_num = n;
                               clear(c_set[x]);
                               cntr[x]- >init = 1; }
```

This macro implements the initialization of a semaphore structure. It consists of a CSIM statement (clear) and regular C code. This macro will be supported by the Ortho-C compiler so actual i860 code will be generated for it. All macros are explicitly invoked by the programmer.

The simulation results can be used to compare the *relative* performance of different algorithms and programming strategies and observe the scalability of OMP. Because the simulator runs on a SUN workstation the execution time estimates for the parallel algorithms cannot be entirely accurate. Simulator users are therefore cautioned to avoid interpreting the results in an *absolute* sense. Two parallel programs have been simulated on a SUN workstation: one for *sorting* and the other for *matrix multiplication*. These two parallel algorithms were originally specified in [20]. The orthogonal sorting algorithm was recursively specified in [20]. What we did in this simulation is to unfold all the recursions to exploit maximum parallelism down to the deepest recursion. The parallel matrix multiplication takes full advantage of OMP in fetching rows or columns of the matrix elements using interleaved access of the orthogonal memory.

Program execution time is estimated by counting the number of simulated RISC instructions. The test programs are first compiled on a (RISC) SUN processor to generate assembly language programs. These programs are then analyzed and instructions counted. Although instructions are counted at compile time, the simulation time is advanced dynamically at run time. Consider how to advance the simulation time of the *for loop* as an example. Before the control enters the loop body, simulation time has to be advanced for initialization of the loop counter and test of the predicate. The actual computation is carried out in the loop body, and the simulation time is based upon the counted number of RISC instructions in the loop body. After incrementing the loop counter and the next predicate test, time will be advanced again. The total time depends upon the number of iterations performed. Ortho-C language constructs are timed in a similar manner.

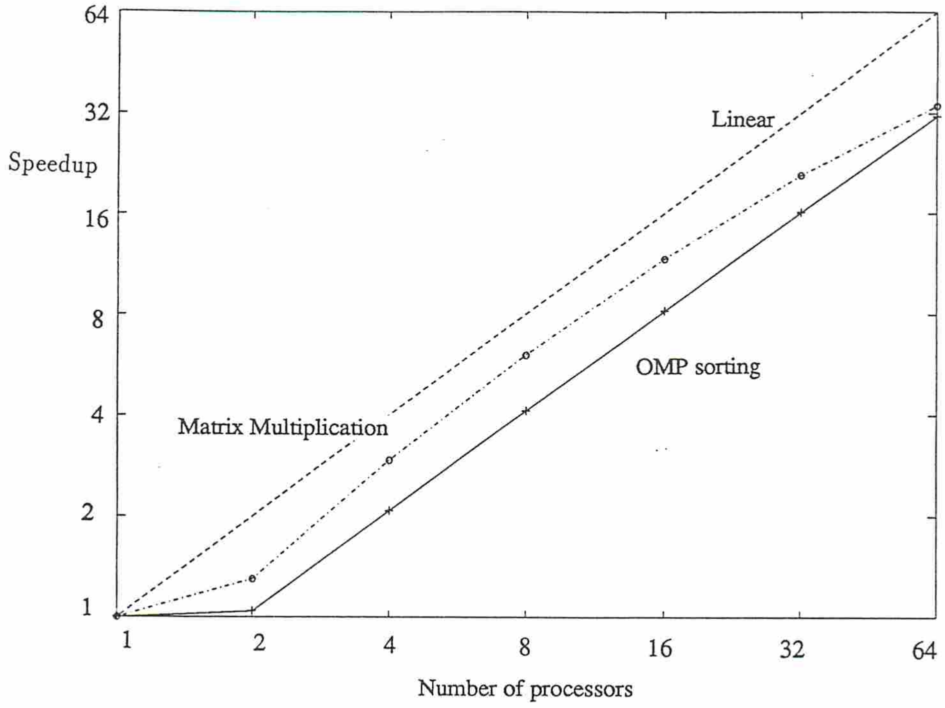
The parallel programs are coded in such a way that each simulated processor gets the same copy of the compiled program. The behavior of the program depends upon which processor it runs on. The primitive, *getpn( )*, is used by a user program to determine the processor number. The portion of the data set that a processor manipulates is indexed by the processor number. The scoping rule is the same as in conventional C language except that global data has to be explicitly assigned to MMs. The performance of the simulated algorithms is heavily dependent upon the data allocation scheme used. These benchmark experiments took advantage of the i860 pipeline instructions. In an *interleaved read*, an array of data is brought to the processor. The matrix multiplication program is coded to exploit this feature by moving the data items in each row or column from the orthogonal memory to a PB buffer first and then utilizing the buffered data.

Before collecting performance data, the machine-dependent overhead should be estimated accurately. This overhead was obtained by running different sizes of data set for the same algorithm on a given size of OMP and taking the average of all the overhead observed. The *speedup* is defined as $S = (T_1 + C_1)/(T_p + C_p)$, where T_p and T_1, are the respective execution times on a p-processor OMP and on a uniprocessor OMP with only local memory and no orthogonal memory, and C_p and C_1 the corresponding overheads.

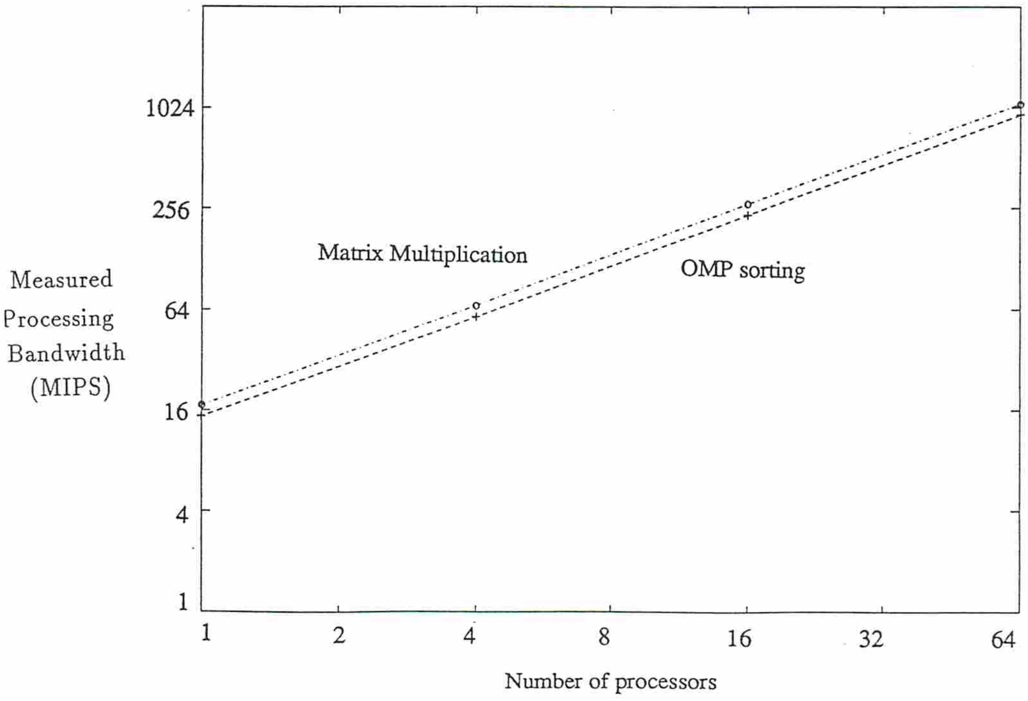
The performance results of the sorting and matrix multiplication algorithms are summarized in Table 5 as well as in Fig.15. Two problem sizes were experimented upon. In the case of sorting, we sorted an array of n randomly-generated numbers, where $n = 4K$ and $16K$ in 8 simulation runs on 7 machine sizes from 1 processor to 64 processors. Sublinear speedups were observed and are displayed in Fig.15. This implies that all the processors in the simulated OMP were almost fully utilized and maximum parallelization was exploited

Table 5. Performance of Sorting and Matrix Multiplication on Simulated OMP Computers of Various Sizes

Machine Size		1 Processor		4 Processors		16 Processors		64 Processors	
Algorithm	Problem Size	Time(sec)	MIPS Rate	Time(sec)	MIPS Rate	Time(sec)	MIPS Rate	Time(sec)	MIPS Rate
		Speedup		Speedup		Speedup		Speedup	
Sorting	64x64	0.48	14.52	0.23	56.71	0.06	227.03	0.02	909.50
		1.00		2.06		7.97		28.28	
	128x128	3.24	14.76	1.57	57.62	0.40	231.94	0.10	923.36
		1.00		2.07		8.16		31.29	
Matrix Multiplication	64x64	0.62	14.89	0.21	58.90	0.07	235.34	0.02	935.30
		1.00		2.90		9.20		25.00	
	128x128	4.69	14.89	1.60	58.88	0.40	235.86	0.14	938.94
		1.00		2.93		11.69		33.70	



(a) Speedups



(b) MIPS Rates

Fig. 15 Speedup and bandwidth performance of the simulated OMP with various machine sizes.

with small overhead time introduced.

As revealed in Fig.15, the speedup and MIPS-rate curves do increase steadily with the increase of machine sizes. Due to the fact that regularly structured parallelism is embedded in dot-product operations, the matrix multiplication achieves higher speedup than the sorting algorithm. The increase in problem size from 64×64 to 128×128 data elements shows only slight improvement. However, when the machine size grows to 64, both algorithms approach a speedup around 32, which is about 50% efficiency observed. In the 16-processor case, the speedup ranges from 7.97 to 11.69 with an efficiency between 50% and 73%, depending on the problem size.

We have also measured the MIPS rate of the simulated OMP for various machine sizes. For the 16-processor OMP, the measured performance is around 227 MIPS in running the sorting algorithm, and 235 MIPS in running the matrix multiplication algorithm. Comparing with the 301 MIPS calculated from using Eq.1 in Section 2, we achieved 75% to 78% of the calculted peak performance. The performance data from these two simulated algorithms on OMP provide a check point of the processing bandwidth given in Eq.1. We are encouraged with the simulation results, which essentailly validated the accuracy of the design parameters chosen.

It should be noted that the simulation results were obtained with a very conservative design based on using 33 MHz i860 chips and the worst case memory latencies calculated at various levels. Once we enter the implementation phase of the OMP project, these parameters will be further refined. For example, using 40 MHz i860 chips and better packaging technology, we can further shorten the memory latencies and synchronization overhead assumed. This imples that the projected 300 MIPS performance for the 16-processor prototype could be further upgraded. Continued simulation runs on other parallel algorithms will be reported based on those refined system parameters. Of course, once the system is completely constructed in 1991, we will report the real benchmark results.

7. Conclusions

We have reported the board-level OMP architecture design of the processors, memories, system interconnects, and host interfaces; the embeddings and mappings of other parallel architectures onto OMP; and the initial performance data generated from the CSIM-based OMP simulation experiments. What we can conclude at this point is that the OMP does support scalable performance and modular expansion, which are our primary goals. Many algorithms which were originally developed for the *mesh*, *hypercube*, and *pyramid* computers, are shown readily convertible to run on the OMP system. Our experimental system supports either synchronized SIMD or asynchronous MIMD operations under direct hardware control and modified Mach/OS support.

Other key contributions of OMP project, besides delivering scalable performance, include the system reconfigurability between SIMD and MIMD modes; better match between multiprocessor bandwidth and that of the orthogonal memory; expandibility to support massive parallelism by increasing the dimension or the multiplicity of the OMP(n, k) architecture; and the use of orthogonal memory and spanning buses for orthogonal vector data communications. We emphasize the mechanisms of global data broadcast, exchange and permutation in any row, column, or diagonal of matrix elements, which are often needed in scientific computations. The OMP prototype will be used as a research vehicle for the Viscom Project at USC. Research and development progress made in the areas of hardware, software, programming, and benchmarking will be reported in a *Series of Technical Reports* published by USC's Laboratory for Parallel and Distributed Computing.

Acknowledgements: The research team of our USC/NSF Viscom Project is divided into three groups focusing on *architecture, software*, and *applications,* respectively. This report is generated by the architecture group. We would like to thank the scientific input provided by individuals from the software and application groups. In particular, we appreciate the interactions with Ellis Horowitz, Rama Chellappa, Bing Sheu, Navid Haddadi, Weihua Mao, Viji Balan, Chi-Wai She, Halima El Naga, and Mark Lytwyn during group meetings and technical discussions. We like to acknowledge the financial support from NSF under grant No. MIP-89-04172 and from USC's School of Engineering in terms of cost sharing and laboratory renovation. Technical assistance from Alliant Computer Systems, Intel Corporation, AST Research, Sun Microsystems, Texas Instruments, Supercomputer Systems, and AMAX Applied Technology is also highly appreciated.

References

[1] N. Ahuja and S. Swamy. Multiprocessor Pyramid Architectures for Bottom-Up Image Analysis. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 6(4):463–474, 1984.

[2] N. A. Alexandridis. Architectural Adaptations in Image Processing Supersystems. In *Proc. First International Conference on Supercomputing Systems*, pages 173–181, 1985.

[3] Alliant and Intel. New Parallel Computing Standards for the i860 Architecture. News Release, 16 October 1989.

[4] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, and M. Lam. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Transaction on Computer*, C-36(12), Dec 1987.

[5] J. L. Baer and W. H. Wang. Architectural Choices for Multi-Level Cache Hierarchies. In *Proc. 16th International Conf. on Parallel Processing*, pages 258–261, Aug 1987.

[6] L.N. Bhuyan and D.P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Transcations on Computers*, C–33(4):323–333, April 1984.

[7] F. A. Briggs and M. Dubois. Effectiveness of Private Caches in Multiprocessor Systems with Parallel-Pipelined Memories. *IEEE Transaction on Computers*, C-32(1):48–59, Jan 1983.

[8] F. A. Briggs, M. Dubois, and K. Hwang. Throughput Analysis and Configuration Design of a Shared-Memory Multiprocessor Systems: PUMPS. In *Proc. 8th Annual Symposium on Computer Architecture*, pages 67–80, May 1981.

[9] P. Budnik and D.J. Kuck. The Organization and Use of Parallel Memories. *IEEE Transaction on Computers*, C–20:1566–1569, 1971.

[10] R.E. Buehrer et al. The ETH Multiprocessor EMPRESS: A Dynamically Reconfigurable MIMD System. *IEEE Transcations on Computers*, C–31(11):1035–1044, November 1982.

[11] R. J. Eickenmeyer and J. H. Patel. Performance Evaluation of On-Chip Register and Cache Organizations. In *Proc. of 15th Symp. Comp. Arch.*, pages 64–72, 1988.

[12] ETA Systems, Inc, St. Paul, Minnesota. *ETA–10 Multiprocessing Simulator User's Guide*, May 1986.

[13] Gateway Design Automation Corporation, Lowell, MA. *Verilog-XL Ref. Manual*, 1989.

[14] J. Goodman and P. J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proc. of 15th Symp. on Comp. Arch.*, pages 422–431, Jun 1988.

[15] A. Gottieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer Designing an MIMD Shared Memory Parallel Computer. *IEEE Transaction on Computers*, C-32(2):175–189, Feb 1983.

[16] N. Haddadi, K. Hwang, and R. Chellappa. Viscom: An Orthogonal Multiprocessor for Early Vision and Neural Computing. In *10th International Conference on Pattern Recognition*, Atlantic City, New Jersey, June 17–21 1990.

[17] D.T. Harper III and J.R. Jump. Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme. *IEEE Transaction on Computers*, C–36:1440–1449, 1987.

[18] K. Hwang and D. DeGroot(eds.). *Parallel Processing for Supercomputers and Artificial Intelligence.* McGraw Hill, N. Y., Mar 1989.

[19] K. Hwang and D. Kim. Generalization of Orthogonal Multiprocessor for Massively Parallel Comppputations. In *Proceedings of the Conference on Frontiers of Massively Parallel Computations*, Fairfax, Virginia, October 10–12 1988.

[20] K. Hwang, P.S. Tseng, and D. Kim. An Orthogonal Multiprocessor for Parallel Scientific Computations. *IEEE Transcations on Computers*, C–38(1):47–61, January 1989.

[21] W. W. Hwu and Y. Patt. Check-point Repair fo High-Performance Out-of-order Execution Machines. *IEEE Transaction on Computers*, C-36(12):1496–1514, Dec 1987.

[22] Intel Corporation, Santa Clara, CA. *i860 Development System User's Guide*, 1989.

[23] Intel Corporation. *i860 Programmer's Reference Manual*, April 1989.

[24] S. Lennart Johnson. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing*, 4(2), April 1987.

[25] D.H. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transaction on Computers*, C–24:1145–1155, 1975.

[26] R. Miller and Q. F. Stout. Mesh Computer Algorithms for Computational Geometry. *IEEE Transaction on Computers*, 38(3):321–340, Mar 1989.

[27] R. S. Nikhil and Arvind. Can Dataflow Subsume Von Neumann Computing. In *Proc. of 16th Symposium on Computer Architecture*, pages 262–272, May 1989.

[28] W. Oed and O. Lange. On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems. *IEEE Transaction on Computers*, C–34:949–957, 1985.

[29] C. D. Polychronopoulos and D. Kuck. Guided Self-Scheduling Scheme for Parallel Supercomputers. *IEEE Transcation on Computers*, C-36(12):1425–1431, Dec 1987.

[30] C. H. Russell and P. J. Waterman. Variations on Unix for Parallel-Processing Computers. *Communication of ACM*, 30(12):1048–1055, Dec 1987.

[31] I.D. Scherson and Y. Ma. Analysis and Applications of the Orthogonal Access Multiprocessor. *J. for Parallel and Distr. Computing*, 7(2):232–255, October 1989.

[32] H.D. Schwetman. CSIM: A C-Based, Process-Oriented Simulation Language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.

[33] H.D. Schwetman. CSIM Reference Manual (Revision 13). Technical report, Microelectronics and Computer Technology Corporation, Austin, Texas, January 1989.

[34] H. J. Siegel, R. J. Mcmillen, and P. T. Muller. A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems. In *AFIPS Conf. Proc.*, 1979.

[35] H. J. Siegel, T. Schwederski, N. P. Davis, and J. Kuehn. PASM: A Reconfigurable Parallel System for Image Processing. In *Proc. of 1984 Workshop on Algorithm-Guided Parallel Architectures for Automatic Target Recognition*, July 1984.

[36] D. Tabak. High-Performance RISC Systems. *J. of Microproc. and Microsys.*, August 1989.

[37] S. L. Tanimoto. A Hierarchical Cellular Logic for Pyramid Computers. *Journal of Parallel and Distributed Computing*, 1:105–132, 1984.

[38] Viewlogic Systems, Inc., Marlboro, MA. *Workview Ref. Manual*, 1989.

[39] D. C. Winsor and T. N. Mudge. Analysis of Bus Hierarchies for Multiprocessors. In *Proc. of 15th Symposium on Computer Architecture*, pages 100–107, Jun 1988.

[40] L.D. Wittie. Communication Structures for Large Networks of Microcomputers. *IEEE Transcations on Computers*, C–30(12):273–284, December 1981.

[41] P. C. Yew. Architecture of the Cedar Parallel Supercomputer. In G. Paul and Eds G. S. Almasi, editors, *Proc. of the 1986 IBM Europe Institute Seminar on Parallel Computing*. North-Holland, Amsterdam, 1988.

Appendix

Pseudo i860 Code for Vector Multiply2 : $C(m \times n) = A(m \times 1)\ X\ B(1 \times n)$

```
    begin
      Load A into cache ( a Vector Load from orthogonal memory )
      Load B into cache
      $i := 0; j := 0;$
      Initiate dual mode operation of i860
  Loop:
        d.fnop ( dummy dual mode floating point instr.)
          load $B[j]..B[j+4]$ from cache to i860 registers ( 2nd instr. of dual pair )
        d.Multiply $A[i] * B[j]$ : dest0 ( dual mode floating point instr. )
          store dest0 register into $C[i, j-3]$ ( 2nd instruction of dual pair )
        d.Multiply $A[i] * B[j+1]$ : dest1 ( dual mode floating point instr. )
          store dest1 register into $C[i, j-2]$ ( 2nd instruction of dual pair )
        d.Multiply $A[i] * B[j+2]$ : dest2
          store dest2 register into $C[i, j-1]$
        d.Multiply $A[i] * B[j+3]$ : dest4
          store dest4 register into $C[i, j]$
        If not end of B then begin $j := j + 4$; go to LOOP; end
        elseif not end of A then
          begin
            $i := i + 1; j := 0;$
            go to Loop
          end
        else
          begin
            flush pipeline
            store trailing values to $C[m-1, n-3], C[m-1, n-2]$ and $C[m-1, n-1]$
            exit dual mode
          end
    end.
```