# Performance Analysis of A Simulated Orthogonal Multiprocessor[1]

Kai Hwang and Chien-Ming Cheng

Technical Report 91-08

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562 (213)740-4470
hwang@panda.usc.edu; cheng@pollux.usc.edu

March 1, 1991

[1]Accepted to appear in The Journal of Parallel and Distributed Computing, Vol. 12, No. 2., July, 1991.  This research is supported by National Science Foundation Grant No. MIP89-04172 to the University of Southern California.

# Performance Analysis of A Simulated Orthogonal Multiprocessor*

Kai Hwang and Chien-Ming Cheng

University of Southern California
Department of Electrical Engineering-Systems
Los Angeles, CA 90089-1115, U.S.A.

Abstract:

This paper analyzes the simulated performance of the *Orthogonal Multiprocessor* (OMP) under development at the University of Southern California. The OMP is simulated with hardware parameters used in the system design. The results are obtained by using a CSIM-based multiprocessor simulator developed at USC. The OMP was evaluated in SPMD (*Single Program and Multiple Data streams*) mode, which demands interprocessor synchronizations at the subprogram level, rather than at instruction lockstep level as in an SIMD machine. The simulated OMP consists of 16 Intel i860 processors and 256 memory modules interconnected by 2-dimensional spanning buses. The simulated benchmarks include unfolded matrix multiplication, 2-dimensional FFT, orthogonal sorting, and parallel pattern clustering. These simulation experiments resulted in a speedup factor between 10 and 15, as compared with a system using a single i860 processor. The paper also reports program and data partitioning methods for the OMP and discusses the effects of synchronization and vector register windows on the multiprocessor performance.

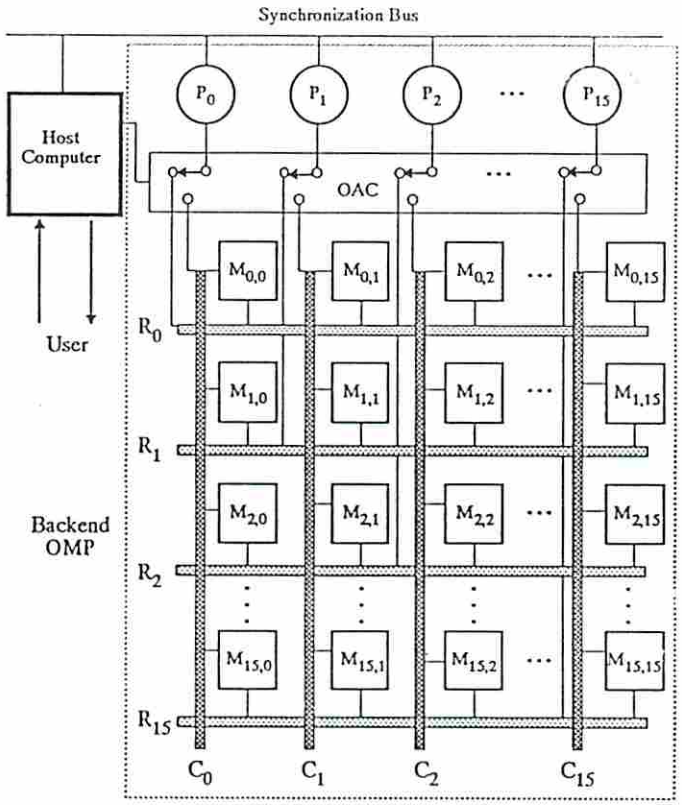
* Accepted to appear in The *Journal of Parallel and Distributed Computing*, Vol. 12, No. 3, July 1991.
 This research was supported by NSF Grant No. 89-04172.

# 1 Introduction

With rapid advent in VLSI technology, 64-bit microprocessors are now equipped with large program/data cache, memory management unit, floating-point accelerators, and hardware graphics on a single CMOS chip. These processors support superscalar, RISC, or VLIW operations resulting in multiple instructions executed per cycle. The Intel i860, and IBM RISC System/6000 are good examples [13]. On the memory side, the access latency has been greatly reduced almost matching the processor cycle time. Carefully designed multiprocessors must be equipped with an efficient memory hierarchy and I/O facilities [6]. The USC *Orthogonal MultiProcessor* (OMP) [7, 10] is an attempt to build such a system using state-of-the-art RISC processors, orthogonal-access memories, spanning buses, and programming tools to support parallel processing. The OMP architecture is well suited for scientific applications regularly structured with parallelism. Typical such applications include matrix computations, image processing, computer vision, and artificial neural network simulation [4, 5, 16].

In order to validate the hardware design as well as to create a user friendly programming environment, we have implemented an OMP simulator based upon the CSIM package developed by Schwetman [20]. The OMP simulator is *algorithm-driven* [16], which models parallel computations in *Single Program Multiple Data streams* (SPMD) mode. In this mode, a large program is partitioned into multiple subprograms to be executed by all the processors in parallel. The SPMD model differs from the conventional SIMD model in that the former handles interprocessor synchronization at the subprogram looping level, whereas the latter chooses lockstep operations at the instruction level. The SPMD mode for parallel processing was first characterized by Karp [12] in 1987.

As depicted in Fig.1, a typical program for the OMP consists of a main program running on a *host* and multiple subprograms running on the backend OMP processors. These subprograms are created as medium-grain child processes at the loop level by the main program after compile time. All programs are coded in an extended C language containing a set of concurrent primitives specially tailored for OMP operations. The CSIM simulation package is extended to support these parallel constructs. The host is used to run a cross




$P_i$ : Processor i  $R_i$ : Row Bus i  OAC : Orthogonal Access Control
Mi,j : Memory Module ij  $C_i$ : Column Bus i

Figure 1: Logical architecture of the USC orthogonal multiprocessor.

compiler for parallelization and to download partitioned subprograms and data sets.

The OMP architecture is briefly described in Sec.2 with emphasis on the ordering in orthogonal memory access. Section 3 presents program and data partitioning methods for exploiting parallelism in user programs. Section 4 describes the construction of the CSIM-based OMP simulator as well as the setting in which simulated benchmark experiments were performed. In Sec.5, parallel mappings of benchmark algorithms are presented with illustrations. New mapping methods are developed to use the *Vector Register Windows* (VRWs) [17] and the *orthogonal-access memory* [7, 19]. Benchmark algorithms simulated on the OMP include unfolded matrix multiplication, orthogonal sorting, 2-D FFT, and parallel pattern clustering.

Section 6 presents several analytical functions to model the performance of OMP with emphasis on SPMD operations. Analytical performance bounds are obtained for the four benchmark programs. We focus on the performance metrics of *speedup factor* and *Million-Operations-Per-Second* (MOPS). The MOPS performance includes both RISC integer operations and floating-point operations as both can be executed concurrently in an i860 processor. Simulation results are reported in Sec.7, where the measured performance is compared with theoretical bounds. Finally, we summarize the research contributions and comment on extended applications of the orthogonal multiprocessor.

# 2 The USC Orthogonal Multiprocessor

The board-level architectural design of a 16-processor OMP has been reported in a *1990 ACM Supercomputing Conference* paper [7]. In this section, we briefly specify the system architecture. Our simulation experiments are based upon parameters actually used in the hardware design. The access ordering ensures the correct execution of parallel programs on the OMP.

## *A. System Architectural Specification*

The logical structure of a two-dimensional OMP is depicted in Fig.1. The system is constructed with $n$ *processors* $P_i$ for $i = 0, 1, \ldots, n-1$, and $n^2$ *memory modules* $M_{ij}$

for $i, j = 0, 1, \ldots, n-1$. The memory modules are organized in a two-dimensional array, with each row (column) of memory modules connected by a dedicated row (column) bus. Each processor uses a dedicated pair of row and column buses to access either a row of memory modules, called *row access*, or a column of memory modules, called *column access*. Each memory module $M_{ij}$ is shared by exactly two processors, i.e. $P_i$ accesses $M_{ij}$ in the *row access* mode and $P_j$ accesses $M_{ij}$ in the *column access* mode. The mesh of $n^2$ memory modules form the orthogonal memory.

The unique feature of the OMP architecture lies in its conflict-free orthogonal memory access. In other words, all processors are allowed to access either the row memory modules or the column memory modules at the same cycle. Mixed-mode memory accesses at the same time are prohibited. This mutual exclusion is enforced using either software or hardware control. Software control is achieved in the subprogram level. Programmers are responsible for ensuring that all processors access the memory in the same mode. Access control can also be achieved using the arbitration logic shown in the OAC box of Fig.1.

We have conducted critical timing analysis with simulation experiments. The purpose is to optimize the architectural design with simplified packaging and implementation considerations. Figure 2 shows the board-level design of a 16-processor OMP. Detailed OMP architecture design was reported in [7, 8, 10, 17]. Highlighted below are the key features:

1. Sixteen 64-bit i860 RISC microprocessors were designed to operate at 40 MHz in the backend processors. A SUN Sparc Engine 300E was used as the frontend host. Host and backend processors communicate over a VME bus.

2. There are 4M bytes of local memory and 512K bytes of vector register windows per each processor. The system has 256 orthogonal memory modules with a total capacity of 256M bytes in the orthogonal memory.

3. We decided to customize the Mach/OS for our purposes. The OS runs in the host as well as in the i860 backend processors. Special C language extensions were developed for scientific and imaging applications.

4. The OMP is targeted to achieve a peak performance of 400 RISC integer MIPS and 640

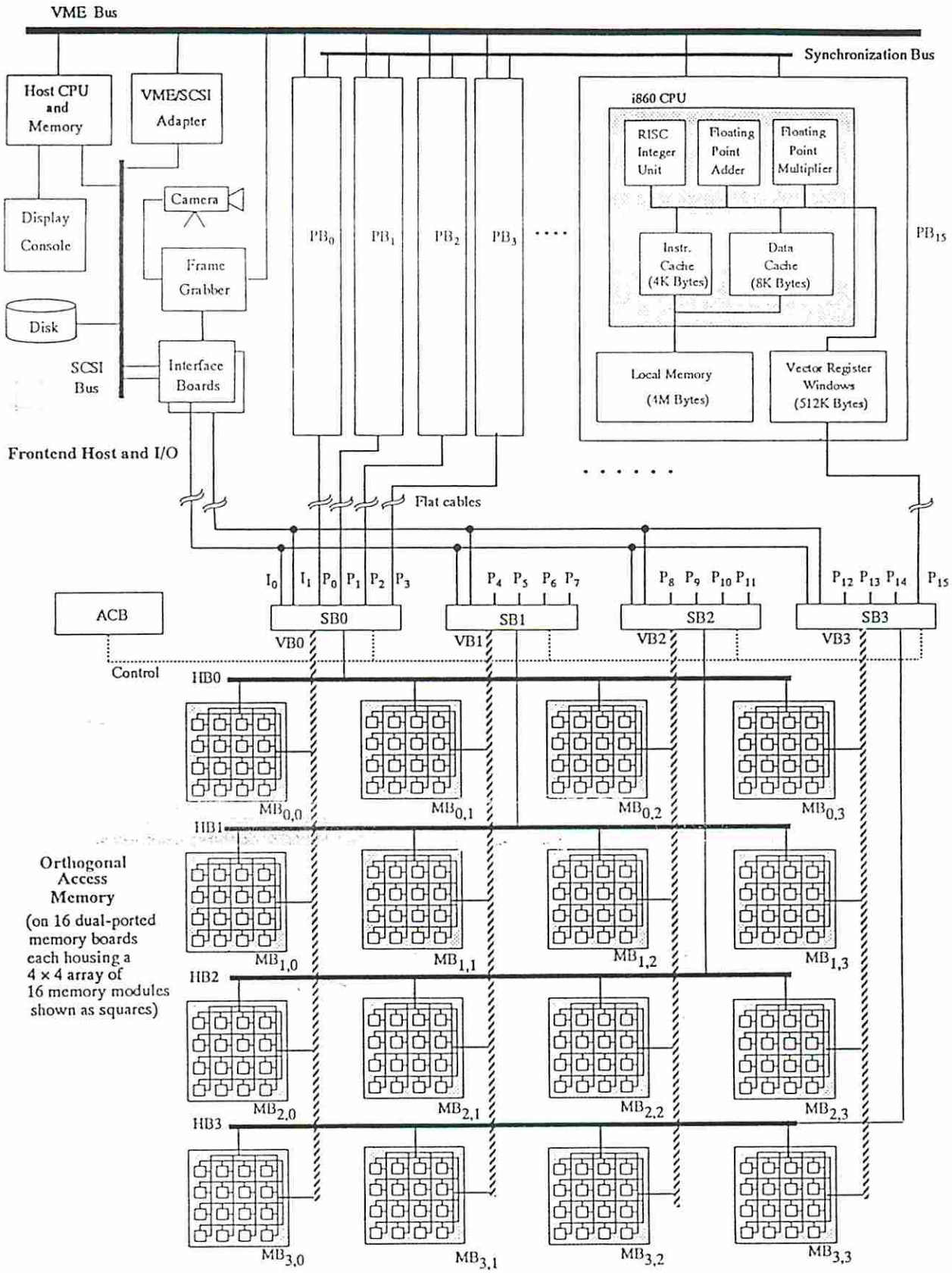



Figure 2: Board level implementation of the orthogonal multiprocessor with sixteen i860 processors and 256 memory modules interconnected by 8 spanning buses. ($PB_i$: Processor boards, $MB_{i,j}$: Memory Boards, SBi: Switch Boards, ACB: Access Control Board, HBi: Horizontal Bus, VBi: Vertical Bus)

Mflops with a combined throughput rate of 550 MOPS. Some of the target performance are met in the simulated benchmark experiments.

## *B. Ordering in Orthogonal Memory Access*

In using the orthogonal memory, we relax the *sequential consistency* [15] constraint to some extent. Basically, the memory hierarchy of the OMP contains the on-board memory and the orthogonal memory as shown in Fig.2. The on-board memory includes the data cache, the instruction cache, the VRWs, and the local memory. The local memory is used to store partitioned program codes and some synchronization variables which may be globally shared by all processors through the VME bus.

The orthogonal memory is used to store shared writable data sets. The mutual exclusion of the orthogonal memory access insures that each processor has its exclusive address space. The vector register windows are a large set of programmable vector registers for buffering and manipulating vector data from the orthogonal memory [17]. All arithmetic/logic operations on vector data are mapped to the orthogonal memory via the VRWs.

Listed below are orthogonal-access conditions that the OMP hardware must follow in order to support SPMD parallel processing.

- *Condition 1*: Accesses to global synchronization variables are indivisible and ordered by the programmers. For example, synchronization using a barrier counter requires users to initialize the counter first, then all the participating processes may join at the counter. The order is coded in user program and must be enforced by the hardware.

- *Condition 2*: Access to global synchronization variables can only be made, when all previous memory accesses are complete. By forcing all pending memory accesses to be performed before mode switch, further accesses to the same variables by other processors will have the most current values.

- *Condition 3*: Memory accesses between two synchronization points can be executed asynchronously, provided data dependencies are preserved.

- *Condition 4*: A *dirty* (modified since its last load to the VRWs) vector data from the orthogonal memory is flushed before an access mode switch. This condition is related to memory coherence. *Selective flushing* is permitted, which allows users to selectively write vector data to the orthogonal memory before mode switch.

# 3 Program and Data Partitioning Methods

Parallelism in OMP programs is expressed in an extended C with new primitives for concurrency control, multiprocessing operations, and row/column mode access control. We describe below how to restructure a sequential program into a parallel program for the OMP, where data partitioning is also illustrated. Described below are several language extensions designed for the OMP system. The semantics of the language extensions are supported by CSIM. The added primitives and their functionalities are highlighted in Table 1. Details of these primitives can be found in [1, 2]. The use of the above language extensions is exemplified by a parallel program written for the unfolded matrix multiplication specified in Fig.3.

**Table 1: C Language extensions for expressing parallelism in the OMP.**

| Primitives | Functionality |
|---|---|
| *forall* | Used by the host program to invoke the execution of all subprograms in the backend processors. |
| *tset_counter* | Used to test whether a barrier counter has been set. If the tested barrier counter has not been set, it will be set to a value indicating the number of processes that are to be joined later. The calling process does nothing if the counter has been set. |
| *mode_select* | Used by running processes to set their access mode. Allowed access modes include *row access*, *column access*, and *local mode*. In the local mode, memory references are made only to the local memory attached to each processor board. |
| *pipeline_read* & *pipeline_write* | Used to transfer vector data between the orthogonal memory and the vector register windows. |
| *broadcast* | Used by any of the OMP processors to send data to all the other processors at the same time. |
| *synch* | Used by cooperating processes to synchronize at a barrier counter. |

## *A. Program Partitioning*

Program partitioning for the OMP involves both functional partitioning and domain decomposition. Functional partitioning leads to the creation of subprograms. Domain decomposition allocates partitioned data sets in the orthogonal memory. Program flow analysis divides single instruction stream into multiple streams. Data dependence analysis separates data blocks associated with the divided program blocks. Language primitives for synchronization and access mode switches are inserted to ensure the correct execution of SPMD programs as revealed in Fig.3.

Consider the matrix multiplication $C = A \times B$, where $(c_{ij}) = \sum_{k=1}^{N} a_{ik}b_{kj}$. The principle is to restructure the algorithm such that all computations can be distributed evenly among all available processors. Recall that the matrix size is $N$x$N$ and the total number of processors is $n$. Dividing $N$ into $n$ equal partitions, we have the block size $K = N/n$ in each block of the partition; i.e.

$$c_{ij} = \sum_{k=1}^{N} a_{ik}b_{kj} \Rightarrow c_{ij} = \sum_{p=1}^{n}\sum_{k=1}^{N/n} a_{ik}b_{kj} \quad (1)$$

Figure 4 illustrates how the parallel algorithm in Fig.3 is developed through program flow analysis. The following steps are taken:

(1) Allocate data items onto the orthogonal memory. In the example, both matrices A and B contain $8 \times 8$ data elements. Matrix A is allocated in row major; while matrix B is allocated in column major. Since four processors (hence, 16 memory modules) are involved in the computation, each memory module stores four data elements.

(2) All processors switch to a column access mode and start fetching elements from the orthogonal memory. The shaded elements in each column of memory modules are fetched and partial products are generated. For example, processor $P_0$ computes $C_{00}^I$, $C_{22}^I$, $C_{44}^I$, and $C_{66}^I$.

(3) After having computed the partial products, all processors must be synchronized and then switch to a row access mode.

Matrix Multiplication

```
1.     forall processors do {
2.       for row_offset = 0 to K -1 do {
3.         for column_offset = 0 to K -1 do {
4.           mode_select(COLUMN)      ; /* set column access mode     */
5.           tset_counter(barrier)    ; /* initialize barrier counter */
6.           for col = 0 to K - 1 do { /* one vector per iteration    */
7.              pipeline_read(A)      ; /* Read a vector of array A    */
8.              pipeline_read(B)      ; /* Read a vector of array B    */
9.              compute_partial_inner_product();
10.          }
11.          pipeline_write(C)        ; /* write out partial results  */
12.          synch(barrier)           ; /* synchronize */
13.
14.          mode_select(ROW)       ; /* set row access mode        */
15.          tset_counter(barrier)  ; /* initialize barrier counter */
16.          for i = 0 to n -1 do
17.             pipeline_read(C)    ; /* read partial inner product */
18.          compute_C()            ; /* compute final result       */
19.          pipeline_write(C)      ; /* write final result         */
20.          synch(barrier)         ; /* synchronize */
21.        } /* for column_offset */
22.      } /* for row_offset */
23.    } /* of forall */
24.    terminate()                  ; /* deallocate OMP resource */
```

Figure 3: A parallel program for unfolded matrix multiplication on the simulated OMP.

(4) Each processor fetches the partial products computed in column operations and generates the final sum. For example, processor $P_0$ fetches $C_{00}^{I}$, $C_{00}^{II}$, $C_{00}^{III}$, and $C_{00}^{IV}$, and produces the final sum for $C_{00}$. In the mean time, $P_1$, $P_2$, and $P_3$ produce $C_{22}$, $C_{44}$, and $C_{66}$, respectively.

(5) Repeat step (2) through (4) until all results are generated. It should be noted that, in each iteration, more than one partial product can be generated, provided the orthogonal memory is sufficient to hold the intermediate results.

### *B. Allocation of Partitioned Data Sets*

The data allocation is statically done at compile time. For example, an $N \times N$ matrix is partitioned evenly to $n \times n$ memory modules, each with a subarray of $K \times K$ data elements, where $K = N/n$. Figure 5(a) shows the data allocation in row-major for a 4-processor OMP, where $N = 8$, $n = 4$, and $K = 2$. Since each row or each column of memory has four modules, each *pipeline_read* or *pipeline_write* involves four data items. These data items have the same offset within the respective memory modules.

Figure 5(b) and (c) show the physical data allocation. Each isolated subblock represents the memory address space of each processor. The data items with the same $(x,y)$ are stored in the same memory module, with their $z$-coordinates representing the offset within each module. In the *column-access* mode, Fig. 5(b), each *pipeline_read* and *pipeline_write* involves one *vector* of data along the $x$-axis. Each orthogonal memory access fetches one *vector* of data along the $y$-axis in the *row-access* mode, Fig.5(c). $K$ rows or $K$ columns in the original matrix are handled in each physical row or column of the orthogonal memory array, respectively. The two solid arrows in Fig.5(b)(c) show the correspondence.

## 4 The OMP Simulator

Simulation of a large parallel computer involves the modeling of both dynamic and static behaviors. The dynamic aspect refers to the modeling of the workload behavior such as memory access patterns, processor service demand, and so forth. While the static behavior includes the modeling of the hardware components and the utilization of these components. The interactions of the workload model with the hardware model are usually supported by

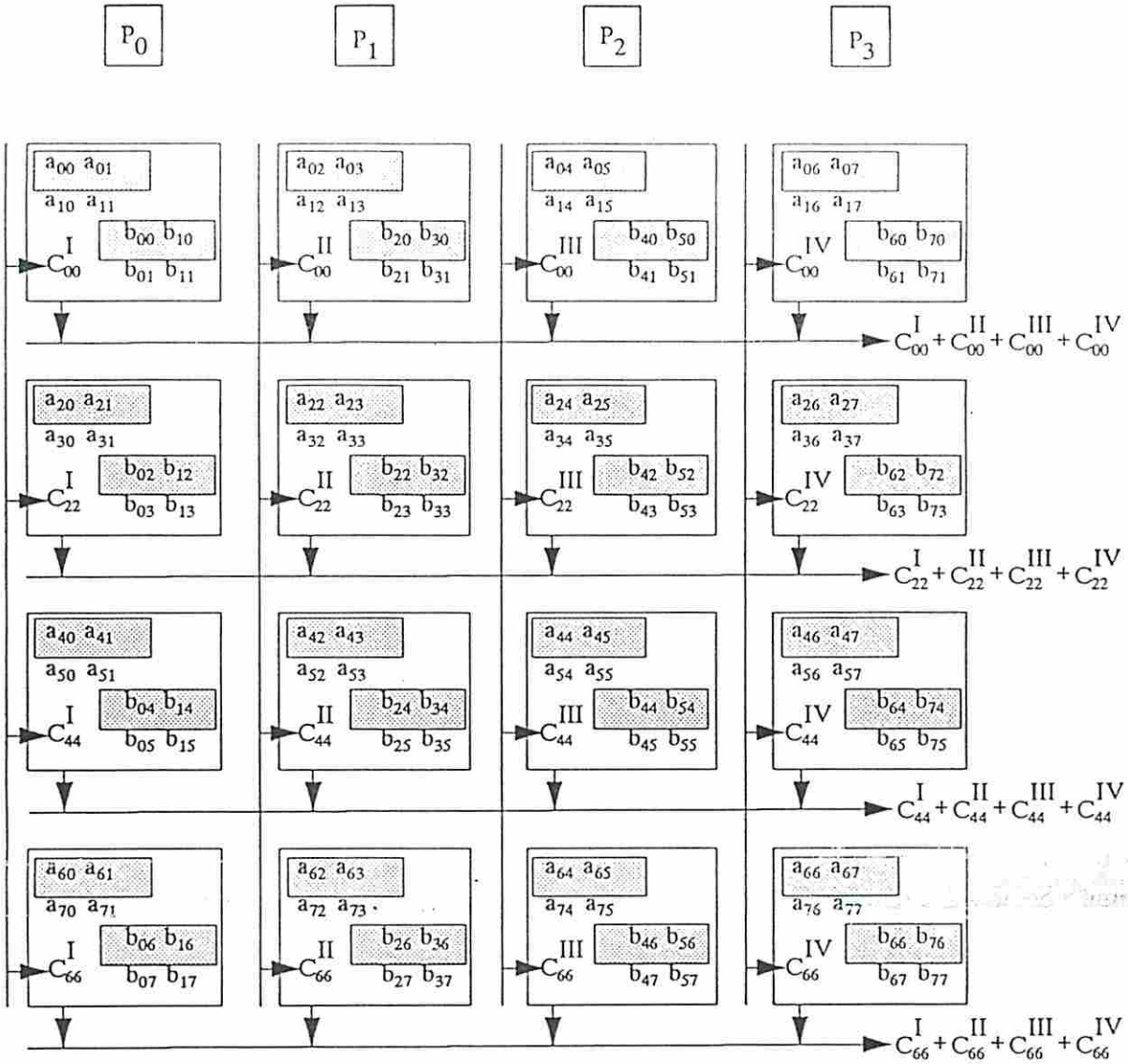



Figure 4: Data partitioning for parallel matrix multiplication on an OMP with 4 processors.

a simulation kernel. This simulation kernel schedules processes, multiplexes real processing time among these processes, and manages event sequencing, etc. In many ways, a simulation kernel mimics the operations of an operating system.

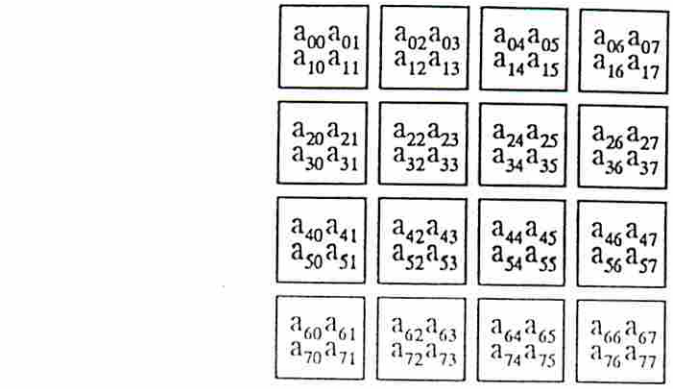
CSIM is a C-based and process-oriented simulation package originally developed by Schwetman [20]. It provides the users with the capability to simulate parallel events in a multiprocessor. *Active processes* (those being executed on the host) use a *hold* statement to cause a specified interval of simulation time to pass. Processes are *suspended*, when the requested resources cannot be granted immediately or when they are waiting for an *event* to occur. The CSIM has been adopted as the simulation kernel of the OMP simulator.

### *A. The Simulator Construction*

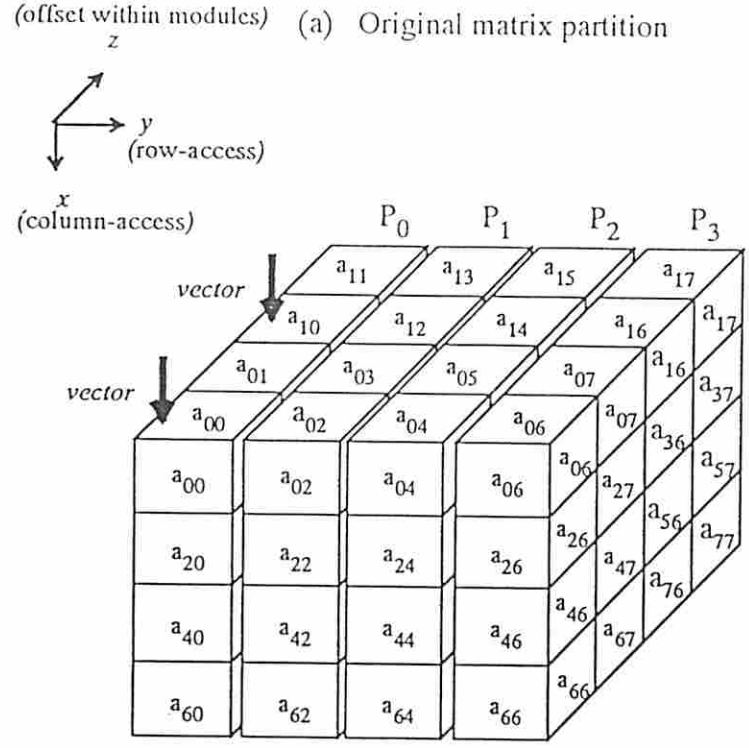
The organization of the OMP simulator is depicted in Fig.6. The OMP simulator is *process-oriented* and *event-driven*. Besides the use of CSIM as the simulation kernel, the development of the OMP simulator has been influenced by the trace-driven concept reported in [3]. The simulator contains a benchmark database, which includes the matrix multiplication, orthogonal sorting, 2-D FFT, and parallel pattern clustering used in our experiments.

The OMP simulator models hardware components by declaring CSIM facilities for them. Modeled components include orthogonal memory, spanning buses, processors, VME bus, and vector register windows. Access to the OMP hardware components is performed through language extensions encapsulating the CSIM *reserve* and *release* statements [20].

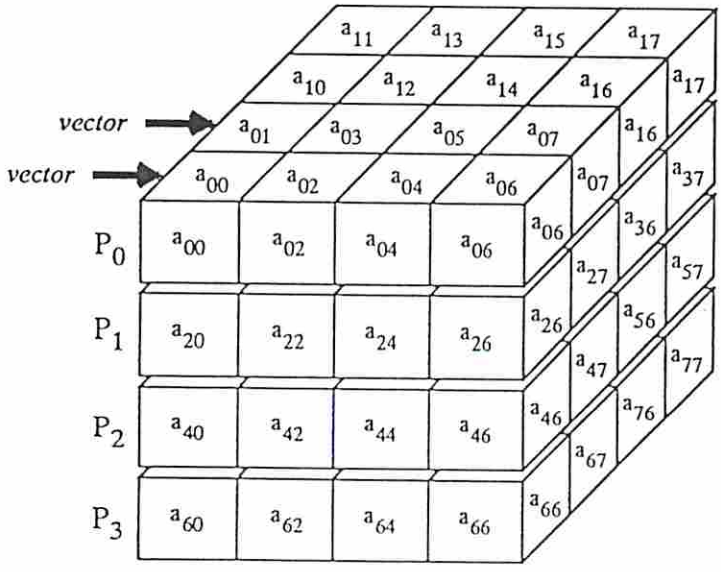
Active OMP processes generate resource requests during the execution of programs. These requests are routed through OMP architecture specifications. Performance statistics are produced and collected while requests are serviced. Our experiments are based on the 40 MHz 16-processor OMP design [7]. The major design parameters used in the simulator construction are listed in Table 2.




(a) Original matrix partition




(b) Storage of matrix elements in the orthogonal memory for column access




(c) Storage of matrix elements in the orthogonal memory for row access

Figure 5: Vectorized access of partitioned matrix elements in the orthogonal-access memory for a 4-processor system.

**Table 2: Hardware Parameters used in the USC Orthogonal Multiprocessor Design (OAM: orthogonal-access memory, VRW: vector register window).**

| Design parameter | Access latency (nsec.) | Comments |
|---|---|---|
| Local Memory Access | 150 | Time to access on-board local memory |
| OAM setup | 825 | Setup time for pipelined access to orthogonal memory. |
| OAM access | 300 | Access time for data in orthogonal memory after the pipeline is setup |
| VME request | 275 | Time for request of VME bus status |
| VME operation | 665 | Time for one *read-modify-write* to VME |
| Instruction execution time | 30 | Based on the calibration value from the i860 development system |
| VRW Access | 75 | Access time of one data item in the vector register window |

## *B. The Simulation Operations*

Unlike trace-driven simulation [14] which demands a mass storage for the derived trace, the algorithm-driven simulation [16] produces and consumes traces on-the-fly. Therefore, it does not introduce any storage overhead. Furthermore, the simulation is accurate to the instruction level. The OMP simulator takes advantage of the multi-thread management scheme of CSIM, which isolates user threads within a UNIX process, and thereby reduces the simulation time significantly.

Code segments in a parallel algorithm are parsed and augmented with event-generating commands. We define *event* as state change of shared resources. Each event-generating command of the OMP simulator encapsulates the CSIM *hold* statement which in turn generates CSIM events. Event sequencing is handled by the CSIM simulation kernel. Besides event sequencing, the computation time of code segments was estimated by instruction counting [22]. As an approximation, the *average execution time* of a code segment was estimated by executing the same code on an i860 development system [11].

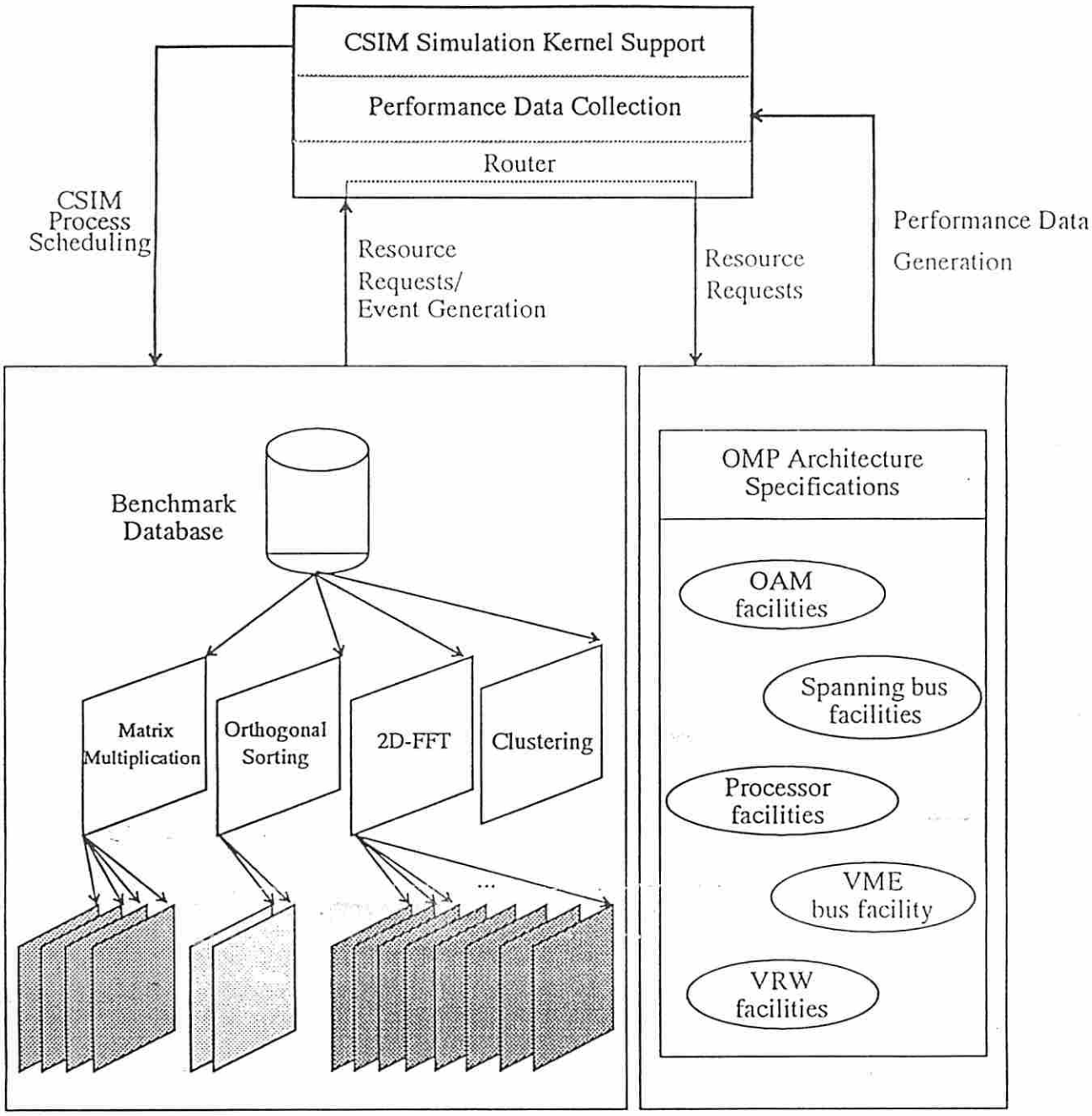



Figure 6: The organization of the OMP simulator using the CSIM simulation kernel for parallel process scheduling and collection of performance data. (OAM: Orthogonal Access Memory, VRW: Vector Register Window).

# 5 Mapping of Benchmark Algorithms

This section describes the mapping of several parallel algorithms onto the OMP. We have developed new mapping schemes to use the VRWs and the 2-dimensional orthogonal memory. The goal is to maximize the resource utilization and to reduce the number of synchronizations. The mappings of four partitioned algorithms are described below. Resource demands, memory reference bounds, and synchronization frequencies in these mappings are revealed in the complexity analysis of computations and communications.

## *A. Unfolded Matrix Multiplication*

The parallel program for matrix multiplication (Fig.3) is analyzed below. The matrix elements, A = $(a_{ij})$ are stored in row major and B = $(b_{ij})$ and C = $(c_{ij})$ are stored in column major. The idea is that all processors compute different partial products simultaneously in alternating column and row access modes. All processors are synchronized, when switching from a column access mode to a row access mode and exchange partial results. These processors then compute the final sum from the collected partial results in a column access mode. In the following complexity analysis, $n$ is the number of processors in an OMP. $N \times N$ is the matrix size, or the number of elements to be sorted, or the number of sample points in an FFT process.

**Lemma 1**

The parallel program specified in Fig.3 for matrix multiplication requires $n+2$ VRWs, $2N^3/n^3 + 2N^2/n + N^2/n^2$ orthogonal memory accesses, and $2N^2/n^2$ synchronizations.

**Proof:**

Let $K = N/n$ be the partitioned block size in mapping a large matrix in the orthogonal memory. Two vectors are needed for operands A and B, pipelined fetched from the orthogonal memory (lines 7-8). These vectors are of size $n$. $n \times n$ partial sums are computed from the two vectors (line 9). Therefore, $n$ vectors are needed for the partial sums. These vectors can be reused for computing the final sums. Therefore, $n + 2$ vectors are needed. The outer two loops (lines 2-3) iterate $K^2$ times. In each iteration, $2K$ *pipeline_read* operations are needed to bring in operands A and B (lines 7-8). Then $n$

*pipeline_write* operations are needed to write the partial results back to the orthogonal memory (line 11). After mode switch (line 14), $n$ *pipeline_read* operations are needed to bring the partial sums from the orthogonal memory to the vector register windows (lines 16-17). One *pipeline_write* is needed to write the final result back to the orthogonal memory. Therefore, $K^2(2K+n+n+1) = 2K^3+2nK^2+K^2$ orthogonal memory accesses are needed. The outer two loops (lines 2-3) iterate $K^2$ times. Each iteration needs two synchronizations. Hence $2K^2$ synchronizations are needed. *Q.E.D.*

## *B. Orthogonal Sorting*

The parallel algorithm for orthogonal sorting is modified from the mesh sorting algorithm reported in [21] for an MIMD multiprocessor, Fig. 7. The first step is to sort data items in each column of memory modules into ascending or descending orders (line 4). These partially sorted subarrays are regarded as bitonic sequences. Then alternating *row_merge* and *column_merge* are performed to merge the entire array into ascending order (lines 8-20).

**Lemma 2**

The orthogonal sorting algorithm requires $N^2/n^2$ VRWs, $4\log nN^2/n^2 + 2N^2/n^2$ orthogonal memory accesses, and $2\log n + 1$ synchronizations.

**Proof:**

Let $K = N/n$ be the block size in partitioning the array being sorted. In order to *sort_one_column*, *row_merge*, and *column_merge*, all the data items in each column or row of memory module are brought to the vector register windows. $K^2$ vectors are needed to store all the data items in one row or column of memory module. Besides one synchronization needed after *sort_one_column* (line 5). The loop (lines 8-20) iterates $\log n$ times. Each iteration requires two synchronizations (line 12 and line 17). Therefore, in total, $2\log n + 1$ synchronizations are needed. Associated with each synchronization, $2K^2$ orthogonal memory accesses are needed, $K^2$ to *pipeline_read* (in line 11) within the *row_merge* subroutine, and the other $K^2$ to *pipeline_write* (line 16) a column or a row of memory modules. Therefore, $(4\log n+2)K^2$ orthogonal memory accesses are needed. The proof is complete by substituting $K$ by $N/n$. *Q.E.D.*

Orthogonal Sorting

```
1.     forall processors do {
2.       tset_counter(barrier) ; /* initialize barrier counter */
3.       mode_select(COLUMN)   ; /* set column access mode     */
4.       sort_one_column(pid)  ; /* sort a column              */
5.       synch(barrier)        ; /* synchronize                */
6.
7.       stride = 1            ; /* stride distance for merging*/
8.       while (array not sorted) {
9.          mode_select(ROW)      ; /* set row access mode          */
10.         tset_counter(barrier) ; /* initialize barrier counter   */
11.         row_merge()           ; /* row merge for the given stride*/
12.         synch(barrier)        ; /* synchronize */
13.
14.         mode_select(COLUMN)   ; /* set column access mode       */
15.         tset_counter(barrier) ; /* initialize barrier counter   */
16.         column_merge()        ; /* col merge for the given stride*/
17.         synch(barrier)        ; /* synchronize */
18.
19.         stride*= 2            ; /* double the stride distance  */
20.      } /* while */
21.   } /* forall */
22.   terminate()                 ; /* deallocate OMP resource */
```

Figure 7: Parallel program for orthogonal sorting on the simulated OMP system.

## C. 2-D Fast Fourier Transform

The parallel program for 2-D FFT is specified in Fig. 8. The idea of performing 2-D FFT on OMP is to perform 1-D FFT along one dimension in the *row-access* mode. All the processors then synchronize, switch to *column-access* mode, and perform another 1-D FFT along the second dimension using the *column-access* mode.

Lemma 3

The two-dimensional FFT requires $N/n$ VRWs, $4N^2/n^2$ orthogonal memory accesses, and one synchronization.

Proof:

In order to perform 1-D FFT on one user-defined data row or column (lines 7 and 17), $N/n$ vectors are needed to store the operands. $2N/n$ orthogonal memory accesses are required, $N/n$ *pipeline_read* to bring data into the vector register windows, and another $N/n$ operations to *pipeline_write* the transformed data back to the orthogonal memory. For each processor, $N/n$ iterations of 1-D FFT are needed along each dimension. Therefore, in total, $4N^2/n^2$ orthogonal memory accesses are needed. It is obvious from the code that only one synchronization is needed (line 21). *Q.E.D.*

The performance of the 2-D FFT program depends on the overhead associated with the orthogonal memory access. For a fixed $N^2$ sample points, a larger OMP performs better, because of the reduced number of orthogonal memory accesses.

## D. Parallel Pattern Clustering

As specified in Fig. 9, the pattern clustering algorithm first selects $L$ samples as the initial cluster centers. This selection is performed by one subprogram. The new cluster centers are then broadcast to all the remaining subprograms. All subprograms classify the patterns in the column access mode. New cluster centers are computed and stored in the orthogonal memory. All subprograms synchronize after they have completed their classification tasks. One subprogram assumes the job of collecting and generating the new cluster centers. These new cluster centers are then broadcast to all remaining subprograms. The procedure of *classify*, *compute_ partial_centers*, *compute_new_centers*, and *broadcast_new_centers*, iter-

2-D Fast-Fourier Transform

```
1.     forall processors do {
2.       tset_counter(barrier) ;  /* initialize barrier counter   */
3.       mode_select(ROW)      ;  /* set row access mode          */
4.       for row = 0 to K - 1 do {  /* fft along K rows of data   */
5.         for row_offset = 0 to K - 1 do /* fetch a row of data  */
6.           pipelined_read(A) ;  /* fetch a vector               */
7.         1D-FFT() ;             /* fft on one row               */
5.         for row_offset = 0 to K - 1 do  /* write a row a data  */
6.           pipelined_write(A) ; /* write a vector               */
8.       }
9.       synch(barrier) ;
10.
12.      tset_counter(barrier) ; /* initialize barrier counter   */
13.      mode_select(COLUMN)   ; /* set column access mode       */
14.      for col = 0 to K - 1 do {/* fft along K columns of data */
15.        for col_offset = 0 to K - 1 do/* fetch a col of data  */
16.          pipelined_read(A) ;  /* fetch a vector              */
17.          1D-FFT() ;            /* fft on one column          */
15.        for col_offset = 0 to K - 1 do /* write a col a data  */
16.          pipelined_write(A) ; /* write a vector              */
17.      }
18.    }
19.    terminate()                ; /* deallocate OMP resource */
```

Figure 8: Parallel program for 2-dimensional FFT on the simulated OMP.

ates, until the cluster centers converge. In the following, we consider a $q$-dimensional feature space consisting of $N^2$ samples to be clustered into $L$ classes by $n$ processors.

Lemma 4

Parallel pattern clustering requires $q + Lq/n$ VRWs. The initialization stage requires $Lq/n$ orthogonal memory accesses, one broadcast, and one synchronization. In the classification stage, each iteration requires one synchronization, one broadcast, and $2qN^2/n^2 + Lq$ orthogonal memory accesses. The computation of the new cluster centers is done by one additional subprogram, which requires $Lq$ orthogonal memory accesses.

**Proof:**

Let $K = N/n$ be the partitioned block size of a large sample space in the orthogonal memory. Although only $P_0$ is involved in the initialization phase, the remaining processors need to wait for $P_0$ to broadcast the result to them. $P_0$ fetches $L$ patterns as the initial cluster centers. $Lq/n$ *pipeline_read* operations are needed. All processors synchronize after the broadcast is completed (line 8 and line 10). These processors then fetch $n$ patterns at a time and perform classification on these patterns, which requires $q + Lq/n$ VRWs.

One synchronization is needed (line 25 or line 27). Only one broadcast is performed by processor $P_0$ (line 24). Since there are $K^2n$ patterns in each column of orthogonal memory modules, $2K^2q$ orthogonal memory accesses are needed to fetch and store all the patterns. Besides, $Lq$ orthogonal memory accesses are needed to place the partially computed cluster centers in the orthogonal memory. $P_0$ needs another $Lq$ *pipeline_read* to bring the partial cluster centers to its VRWs so that the new cluster centers can be computed. The proof is complete by substituting $N/n$ for $K$. *Q.E.D.*

The performance of this clustering program depends upon how the cluster centers are selected. Many iterations may be needed for the program to converge. Within each iteration, only one synchronization is needed. For the same problem size, the number of orthogonal memory accesses decreases as the machine size $n$ increases.

Parallel Pattern Clustering

```
1.    forall processors do {
2.      tset_counter(barrier) ; /* initialize barrier counter    */
3.      if (pn == 0) { /* p0 initialize the feature centers     */
4.        mode_select(COLUMN) ;
5.        fetch_sample_patterns();/* the first L patterns are   */
6.                                /* used as initial centers    */
7.        broadcast(ALL) ;        /* broadcast to all processors*/
8.        synch(barrier) ;
9.      } else { /* other processors wait for p0 to broadcast   */
10.       synch(barrier) ;    /* the initial feature centers    */
11.     } /* end of else */
12.
13.     while (not converge) {
14.       mode_select(COLUMN) ;
15.       classify() ;          /* classify patterns            */
16.       compute_partial_centers();/* compute partially new centers*/
17.       pipelined_write(new partial centers) ;
18.       tset_counter(barrier) ;
19.       if (pn == 0) {
20.         mode_select(ROW) ;
21.         pipelined_read(partial centers) ;
22.                               /* p0 collects partial centers */
23.         compute_new_centers() ; /* and compute new centers  */
24.         broadcast(ALL) ; /* and broadcast to all processors */
25.         synch(barrier) ;
26.       } else {              /* other processors wait for the */
27.         synch(barrier) ;    /* new feature centers from p0   */
28.       } /* end of else */
29.     } /* while */
30.   } /* end of forall */
31.   terminate()                 ; /* deallocate OMP resource */
```

Figure 9: Parallel program for pattern clustering on the simulated OMP.

# 6 Performance Projection

In SPMD operations, the subprograms are compiled in the host and dispatched to the backend processors for parallel execution. At run time, each subprogram is dedicated to a processor. Static resource allocation is assumed at compile time. No process migration is allowed at run time. All subprograms are coordinated by a barrier synchronization mechanism as reported in [9].

The performance of the OMP is projected with three attributes: (1) The numbers of synchronizations and *pipeline_read* or *pipeline_write* operations needed. (2) The number of instructions executed was estimated by running the same program in an i860 development system at USC. The development system provides facility for recording instruction counts. (3) Orthogonal memory access time was estimated from hardware design specifications.

In this section, we show how to use these attributes to project the OMP performance. First, we characterize SPMD program behavior. Then we estimate the speedup and MOPS performance of OMP against 4 benchmark programs.

## *A. SPMD Program Behavior*

An SPMD program executes the same subprogram over different data sets. All subprograms use the same access mode at the same time. The interaction among these subprograms lies in the contention for the shared synchronization bus. We assume the worst case in which any subprogram needs to wait for all the remaining subprograms to reach the synchronization point.

An SPMD program for the OMP basically consists of a mixture of *compute* and *synchronization* operations. The composition of the mixture determines the program behavior, and thus affects the performance. An SPMD program is characterized by a 5-tuple: $\Sigma = (m, \tau, b, n, k)$, where:

(1) $m$ is the total number of instructions to be executed in an SPMD program,

(2) $\tau$ is the *access time* for a barrier counter,

(3) $b$ is the number of compute/synchronize pairs,

(4) $n$ is the number of processors in the OMP, and

(5) $k$ is the number of orthogonal memory accesses, including both *pipeline_read* and *pipeline_write* operations.

Figure 10 shows the execution cycle of an SPMD program with two iterations ($b = 2$). It consists of two pairs of *local computation* and *synchronization*. Note that the local computation time for the two iterations may not be equal. For simplicity, we assume the mean value in our analysis. The number of iterations of the program is application dependent. The *degree of parallelism* corresponds to the number of active subprograms. Jagged edges correspond to subprograms arriving at the barrier counter at different times. Parallelism may decrease as earlier arrivals need to wait for the last subprogram to complete.

## *B. Speedup Performance*

Since the total number of instructions executed is $m$, the number of instructions executed by each subprogram is estimated as $\frac{m}{n}$. Each subprogram is responsible for data movements between the VRWs and the orthogonal memory. Therefore, the total computation time for each subprogram is estimated as $\frac{m}{n}t + k\sigma$, where $t$ and $\sigma$ are the instruction execution time and *pipeline read/write* operation time, respectively.

We assume that the computation time is evenly distributed to all iterations. The local computation time within each iteration is estimated as $\frac{mt}{nb} + \frac{k\sigma}{b}$. The execution time of each iteration is computed as $\frac{mt}{nb} + \frac{k\sigma}{b} + (n-1)\tau$, because the slowest subprogram always keeps the $n-1$ remaining subprograms waiting. The *execution time* ($E$) of an SPMD program is thus estimated as:

$$E = \frac{mt}{n} + k\sigma + (n-1)b\tau \tag{2}$$

The first term is the instruction execution time. The more processors are involved, the less instructions will be executed in each subprogram. The second term reflects the overhead time incurred due to the data transfer between the VRWs and the orthogonal memory. The third term is the barrier synchronization overhead. It is proportional to the number of subprograms involved in the synchronization. When $n = 1$, the second and third

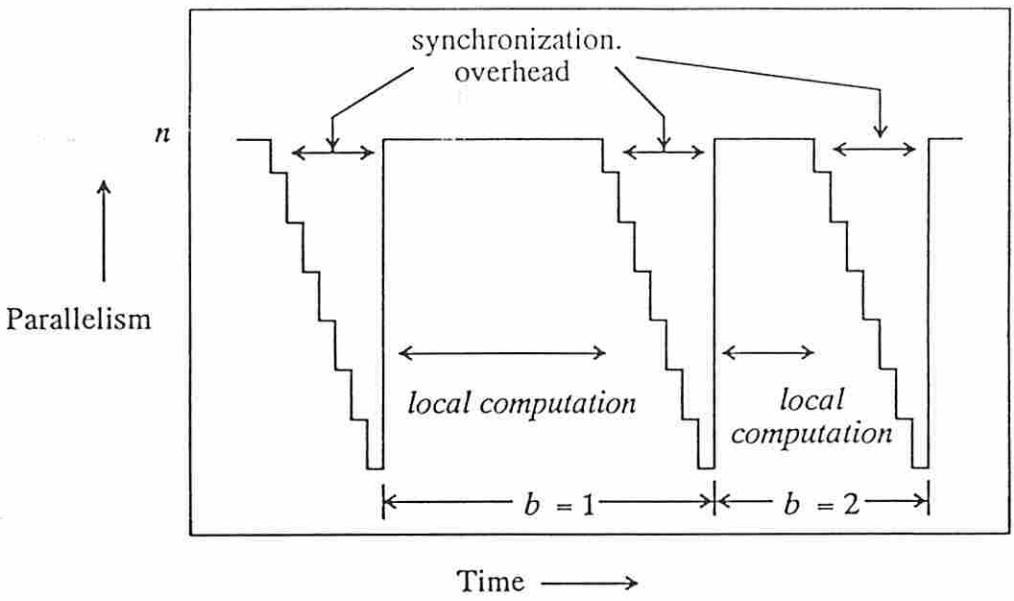



Figure 10: The life cycle of an example SPMD program.

terms disappear, and $E_1 = mt$. A *speedup factor* is defined as $S = E_1/E$. By substituting $E$ and $E_1$ from the above, we obtain the following result:

**Theorem 1**

The speedup resulted from executing an SPMD program on an $n$-processor OMP is given below:

$$S = \frac{mt}{(\frac{mt}{n} + k\sigma + (n-1)b\tau)} \tag{3}$$

as compared with the use of a single processor system for the same purpose.

Sublinear speedup is achievable, when the terms representing synchronization overhead and the orthogonal memory access are much lower than the execution time. For a fixed number of processors, $S$ decreases as the synchronization overhead ($b\tau$) increases. More orthogonal memory accesses will degrade the performance. Substituting the complexity counts in Lemmas 1-4 into Theorem 1, we project the following speedup performance for four simulated benchmark programs:

**Corollary 1**

The speedup factor for parallel matrix multiplication on an OMP is projected as:

$$S = \frac{mtn^3}{(mt + 2\sigma N^2 + 2\tau N^2)n^2 + N^2(\sigma - 2\tau)n + 2N^3\sigma} \tag{4}$$

**Corollary 2**

The speedup factor for the orthogonal sorting on an OMP is projected as:

$$S = \frac{mtn^2}{2\tau n^3 \log n + \tau n^3 - 2\tau n^2 \log n - \tau n^2 + mnt + 4\sigma N^2 \log n + 2N^2\sigma} \tag{5}$$

**Corollary 3**

The speedup factor for performing 2-D FFT on an OMP is projected as:

$$S = \frac{mtn^2}{\tau n^3 - \tau n^2 + mtn + 4\sigma N^2} \tag{6}$$

**Corollary 4**

The speedup factor for parallel pattern clustering on an OMP is projected as:

$$S = \frac{mtn^2}{\gamma\tau n^3 + (2\sigma qL - \tau)\gamma n^2 + mtn + 2\gamma\sigma qN^2} \tag{7}$$

where $\gamma$ is the number of iterations towards convergence.

Recall in Table 2, $t$ was set as 30 nsec. Each *pipeline_read* or *pipeline_write* requires one orthogonal memory setup time and 15 data accesses after the pipeline is set up. Therefore, $\sigma = 825 + 15 \cdot 300 = 5{,}325$ nsec. Each synchronization requires approximately 940 nsec. The total number of instruction executed, $m$, is obtained by running the SPMD program with $n = 1$, and is estimated as 183M, 56M, 87M and 39M respectively for matrix multiplication, orthogonal sorting, 2-D FFT, and parallel pattern clustering. $L$ is chosen as 16. Each pattern is characterized by $q = 3$ features. The number of iterations ($\gamma$) needed to cluster the sample space is equal to 2 in our experiments. By replacing the above parameters into Eq. 4-7, the predicted speedup performance is summarized in Table 3. The *efficiency*, $S/n$, for the four algorithms lies between 75% and 93%.

**Table 3: Projected Speedup of the OMP of Various Sizes**

| Benchmark Algorithm | No. of processors | | | |
|---|---|---|---|---|
| | 2 | 4 | 8 | 16 |
| Matrix Multiply | 0.95 | 3.05 | 7.22 | 15.18 |
| Sort | 1.73 | 3.54 | 7.33 | 15.09 |
| 2D-FFT | 1.87 | 3.87 | 7.87 | 15.86 |
| Clustering | 1.37 | 3.26 | 7.57 | 15.38 |

## *C. MOPS Performance*

The MOPS performance is defined as the total number of instructions executed divided by the execution time. An SPMD program contains three types of instructions count: *$m$ compute instructions, $k$ pipelined access to the orthogonal memory*, and *$b$ synchronization points*. Therefore, the total instruction count is $m + kn + bn$. The total execution time is $\frac{mt}{n} + k\sigma + (n-1)b\tau$. We thus obtain the following throughput estimate:

**Theorem 2**

The MOPS rate of an OMP with $n$ processors is estimated below:

$$T = \frac{m + kn + bn}{\frac{mt}{n} + k\sigma + (n-1)b\tau} \tag{8}$$

In Eq. 8, synchronization overhead and the number of data accesses from the orthogonal memory tend to lower the MOPS rate. By substituting the results from Lemmas 1 to 4 into Eq. 8, we obtain the following estimates of the MOPS rate of the simulated OMP.

**Corollary 5**

The MOPS rate for parallel matrix multiplication on an OMP is projected as:

$$T = \frac{(m+2N^2)n^3 + 3N^2n^2 + 2N^3n}{(mt + 2\sigma N^2 + 2\tau N^2)n^2 + N^2(\sigma - 2\tau)n + 2\sigma N^3} \tag{9}$$

**Corollary 6**

The MOPS rate for orthogonal sorting on an OMP is projected:

$$T = \frac{2n^3 \log n + n^3 + mn^2 + 4N^2 n \log n + 2N^2 n}{2\tau n^3 \log n + \tau n^3 - \tau n^2 \log n - \tau n^2 + mtn + 4\sigma N^2 \log n + 2\sigma N^2} \tag{10}$$

**Corollary 7**

The MOPS rate for performing 2D-FFT on an OMP is projected:

$$T = \frac{(m+1)n^2 + 4N^2 n}{\tau n^3 + mtn + 4\sigma N^2 - \tau n^2} \tag{11}$$

**Corollary 8**

The MOPS rate for parallel pattern clustering on an OMP is projected as:

$$T = \frac{2\gamma q L n^3 + mn^2 + 2\gamma q N^2 n + \gamma}{\gamma\tau n^3 + (2\gamma q \sigma L - \gamma\tau)n^2 + mtn + 2\gamma q \sigma N^2} \tag{12}$$

The projected MOPS rates of the OMP for 4 benchmarks are expressed in terms of the same parameters used to predict the speedup factors. Table 4 summarizes these MOPS rates after substituting the corresponding numerical values. The throughput of a 16-processor OMP is projected to be around 500 MOPS.

**Table 4: Projected MOPS Rate of an OMP of Various Sizes**

| Benchmark Algorithm | No. of processors | | | |
|---|---|---|---|---|
| | 2 | 4 | 8 | 16 |
| Matrix Multiply | 33 | 103 | 242 | 506 |
| Sort | 58 | 118 | 245 | 504 |
| 2D-FFT | 63 | 129 | 262 | 529 |
| Clustering | 46 | 109 | 238 | 498 |

# 7 Simulated Performance Results

This section reports the performance data collected from running four benchmark programs over an OMP simulator. The simulation results are compared with the theoretical projections in Section 6. These simulation results are refined from preliminary results reported in an earlier conference paper [16]. We will also discuss two sensitivity issues on these performance indices.

## *A. Measured Simulation Results*

Figure 11 shows the speedup curves of simulated OMP benchmark programs. These experiments show that sublinear speedup was achieved. For the 16-processor configuration, the speedup factor ranges from 10 for orthogonal sorting to 15 for parallel pattern clustering. The measured speedup is upper bounded by the projected speedups. The computation is distributed equally among all participating processors.

The MOPS performance of the OMP simulation is shown in Fig. 12(a)(b). The MOPS rate is obtained by recording the number of instructions executed in the simulation runs. With 16 processors, the MOPS rates for parallel pattern clustering, 2-D FFT, matrix multiplication, and orthogonal sorting are measured as 516, 455, 450 and 314, respectively. These rates are within 57% to 94% efficiency from the projected peak performance of 550 MOPS.

## *B. Effects of Synchronization Overhead*

Through simulated benchmark experiments, we reveal the effect of synchronization overhead on the MOPS performance. As shown in Fig. 13(a) for the parallel multiplication of two matrices (Fig. 3), the MOPS rate decreases monotonically as the synchronization overhead increases in three problem sizes. The operations performed during busy-waiting period are excluded from the MOPS count. The MOPS rate is very sensitive to the synchronization overhead, when the problem is small in size.

The effect of synchronization frequency on the speedup performance in matrix mul-

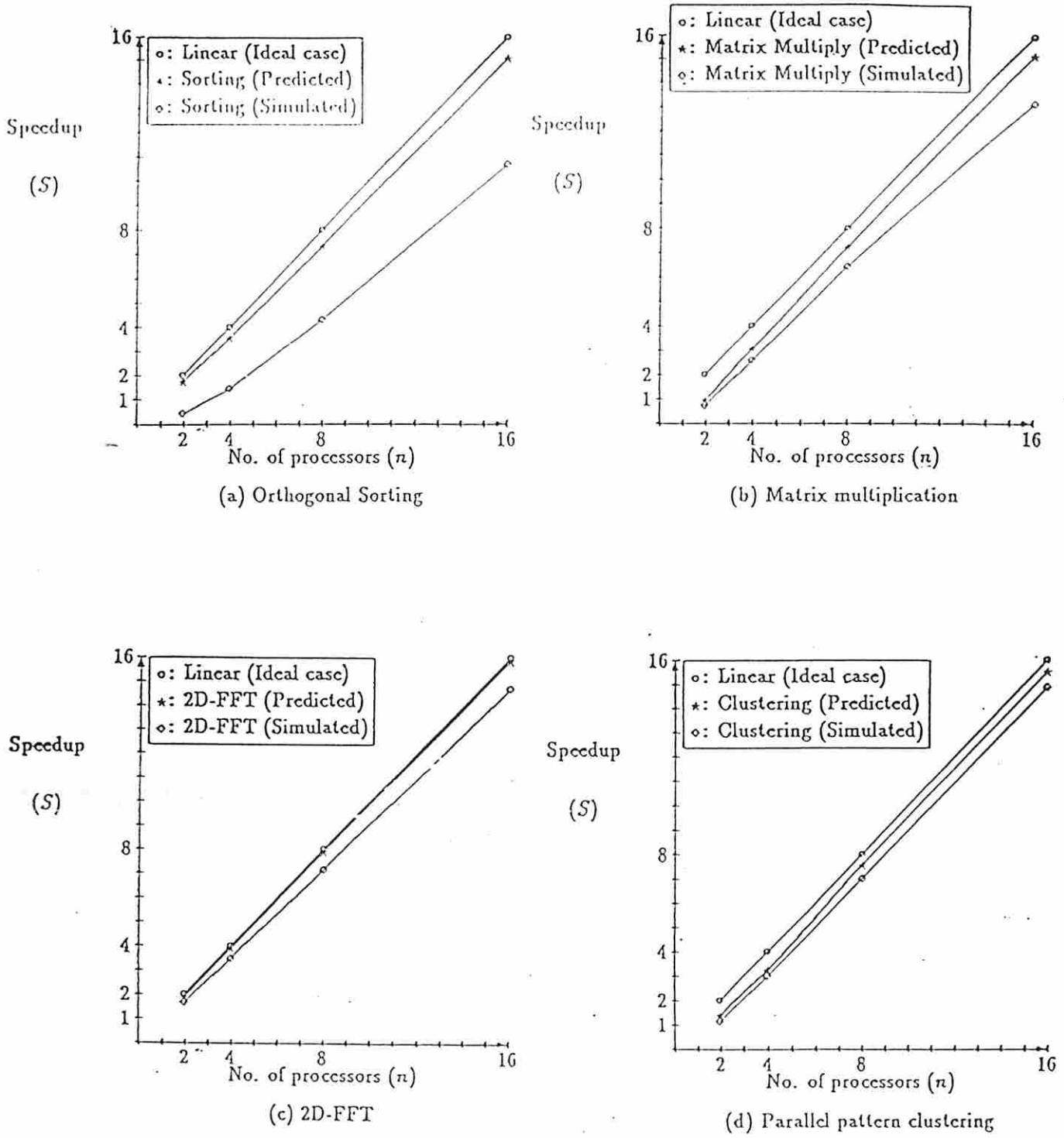
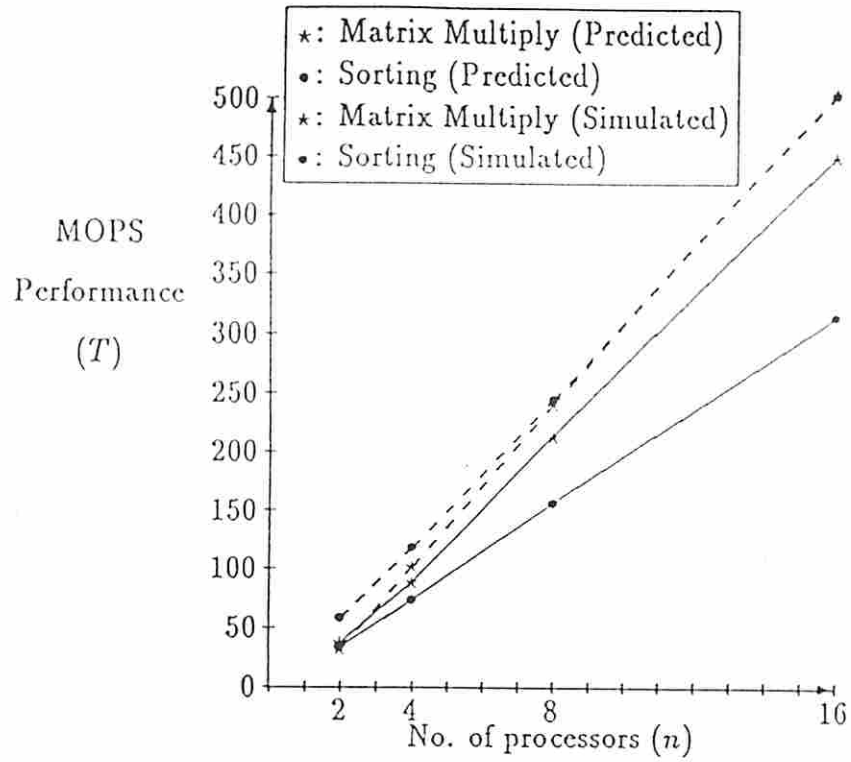
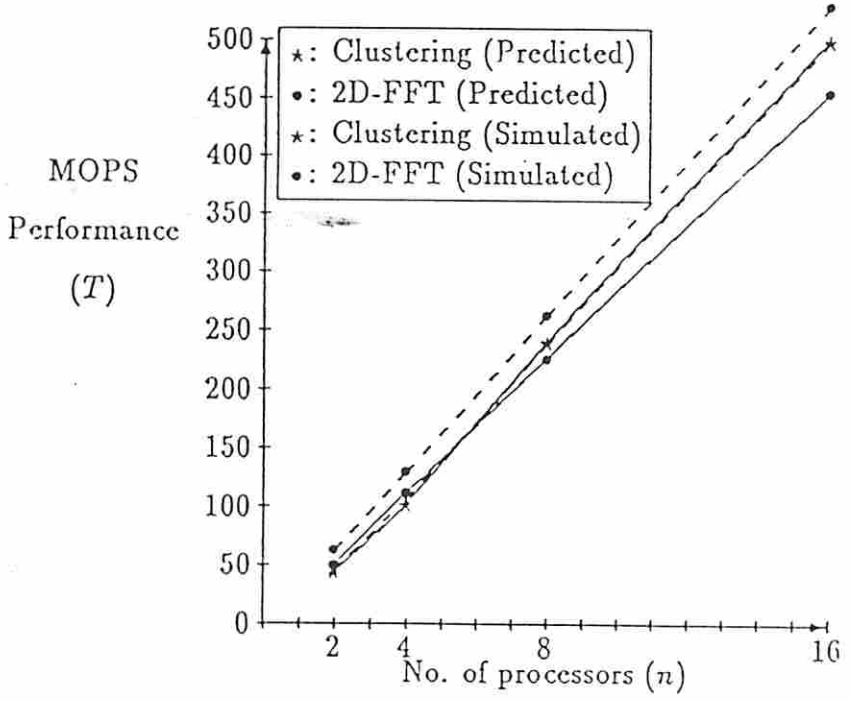



Figure 11: Speedup factors obtained from simulated benchmark experiments on the OMP.

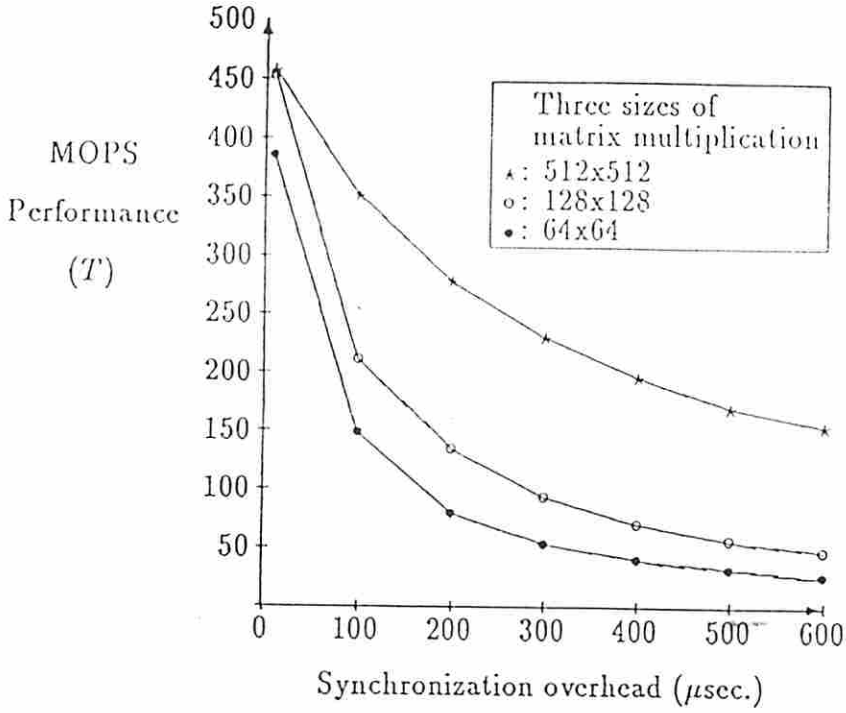



(a) Matrix multiplication and sorting

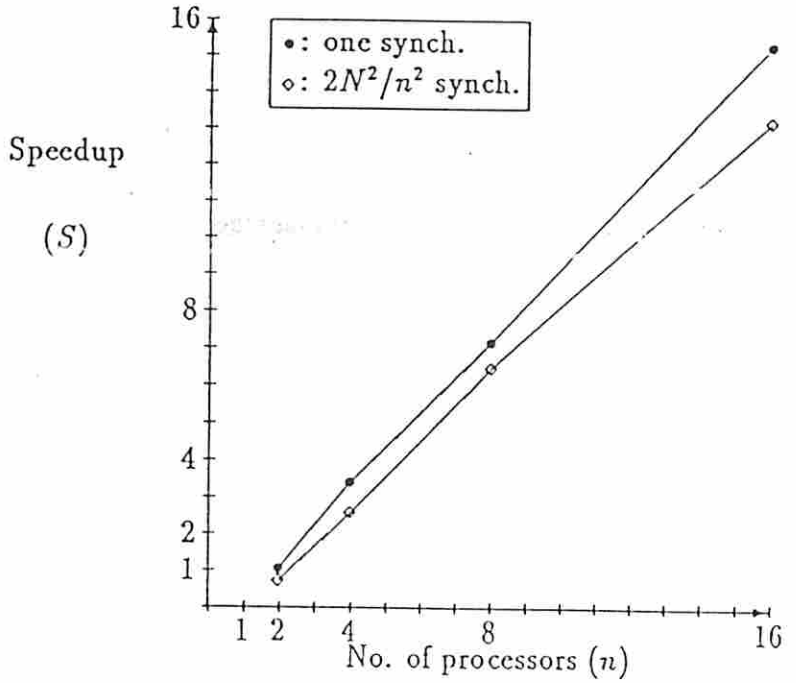



(b) 2-D FFT and pattern clustering

Figure 12: MOPS performance of OMP form the simulated benchmark experiments. (Dashed lines: predicted; Solid lines: simulated)




(a) Synchronization overhead




(b) Synchronization frequency

Figure 13: Synchronization effect on performance of the simulated OMP.

tiplication is shown in Fig. 13(b). The matrix multiplication is coded in two programming styles, one needs $2N^2/n^2$ synchronizations and the other needs only one synchronization. Simulation results show that, for 16-processor configuration, one can achieve a speedup of 15.3, if only one synchronization is needed. Speedup is degraded to 13.2 with the same configuration, when $2N^2/n^2$ synchronizations are needed. More orthogonal memory is needed to store intermediate results, in order to reduce the number of synchronizations.

### *C. Effects of Vector Register Windows*

*Vector register windows* (VRWs) are built on the processor board. They serve as high-speed buffers for vector data transfer between the orthogonal memory and the i860 processors. These windows are user programmable providing various high-level abstractions of matrix data elements. The use of these windows is supported by programming an index memory. This enables an *on-the-fly index manipulation*, while the vector data elements are pipelined into or out of the VRWs.

Fast data movement and index manipulation are supported by VRWs as reported in [18]. The orthogonal memory can be used to realize many interprocessor communication patterns, including *permutation*, *broadcast*, *multicast*, and *arbitrary mappings*. In fact, vectorized memory communication paradigm was introduced to map various multicomputer communication patterns into shared-memory multiprocessors [8].

We evaluate below the effect of *windows size* and *window number* on the OMP performance. When there are sufficiently large number of windows, large-scale matrix multiplications can be sped up by holding all intermediate inner products simultaneously. In fact, there is a tradeoff between the number of windows and the size of each window. The window size is measured as the number of *vector registers* (16 words in each vector and 16 vectors in each vector register) that each window can hold. In Fig.14, we consider the performance (in terms of MOPS rate) of OMP in the multiplication of two 128x128 matrices. Two MOPS curves are obtained: one corresponds to the use of 3 windows and the other of $N^2/n^2 + 2$ windows, assuming equal window size.

It is clear from Fig.14 that the performance is a lot better in using a larger number

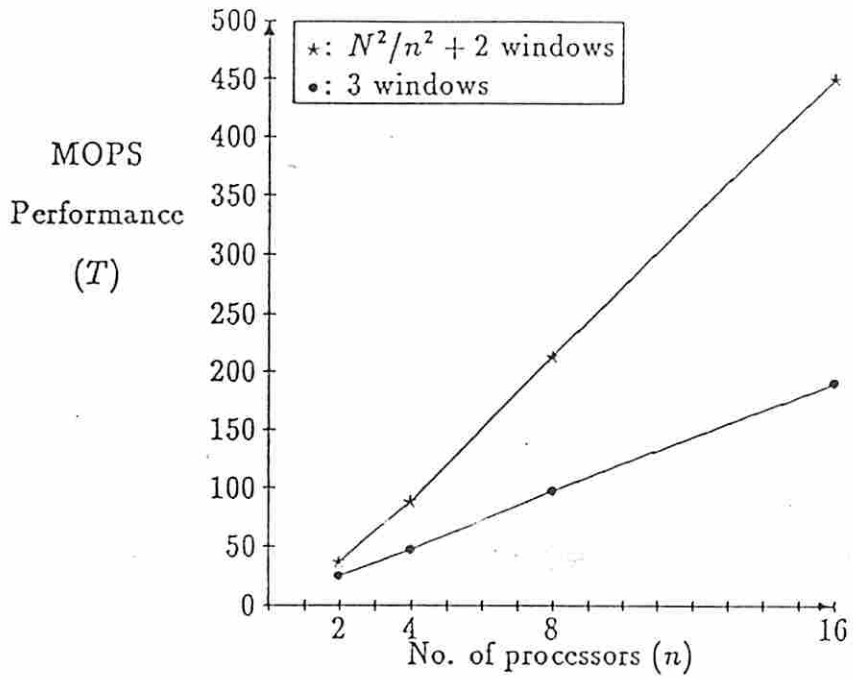



Figure 14: Effect of the window number on the performance of OMP for the multiplication of two 128x128 matrices.

of VRWs, as compared with the use of only 3 windows. This demand on window number is different in image processing or in early vision applications, where optical flow only requires the handling a few image frames at the same time. The tradeoffs between window size and window number are obviously application-dependent. In general, for large-scale matrix computations, more windows lead to better exploitation of parallelism as revealed in Fig.14. However, in image processing of large frames, we want to use larger window sizes instead of more windows.

# 8 Conclusions

We have modeled the OMP performance through multiprocessor simulation and theoretical analysis. The simulated performance is upper bounded by the theoretical prediction. The OMP simulator is proven an effective tool in predicting the OMP performance and in validating the architectural design choices. We have evaluated the use of vector register windows, access latency of the orthogonal memory, and SPMD synchronization overhead.

As illustrated, the orthogonal multiprocessor is capable of exploiting parallelism in scientific supercomputing and image processing applications. The four benchmarked algorithms cover matrix arithmetic and imaging computations. The speedup factors of these benchmarks fall in the range of 10 to 15 for a 16-processor OMP system. A sublinear speedup is therefore achievable. The throughput performance can be as high as 500 MOPS in these benchmark experiments. This implies an efficiency around 91% as compared with the peak performance of 550 MOPS in a 16-processor OMP design.

The performance results imply that the OMP is suitable for structured parallelism in multiprocessors. The larger is the problem size, the greater will be the performance of the OMP. The 16-processor OMP system should be able to perform matrix computations with order $10^5$ and to process 3,500 image frames per second with sustained performance of 500 MOPS or 2.3 $\times 10^8$ pixels/sec. The orthogonal multiprocessor should be encouraged for use in time-critical, large-scale, scientific simulation and image understanding applications.

*Acknowledgments:*

The authors wish to acknowledge the group effort by the USC OMP research team. In particular, Sharad Mehrotra has developed an earlier version of the OMP simulator. The simulation benchmark experiments were assisted by Shisheng Shang and Navid Haddadi. Technical suggestions from Michel Dubois, D.K. Panda, Weihua Mao, Santosh Rao, and Viji Balan are appreciated. The research funding from NSF Grant No. 89-04172 has made it possible to complete this simulation study of the orthogonal multiprocessor.

# References

[1] V. Balan. Trojan-C Language and Its Programming Environment. *Technical Report*, Dept. of Electrical Engineering-Systems, Univ. of Southern California, Los Angeles, March 1990.

[2] C. M. Cheng. Programmer's Guide to the USC Orthogonal Multiprocessor Simulator. *Technical Report*, Dept. of Electrical Engineering-Systems, Univ. of Southern California, Los Angeles, CA, Mar 1990.

[3] M. Dubois, M. Balakrishnan, F.A. Briggs, and I. Patil. Trace-driven Simulations of Parallel and Distributed Algorithms in Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 909–917, August 1986.

[4] N. Haddadi, K. Hwang, and R. Chellappa. Viscom: An Orthogonal Multiprocessor for Early Vision and Neural Computing. In *Proceedings of 10th International Conference on Pattern Recognition*, Atlantic City, N.J., June 1990.

[5] K. Hwang, H.M. Alnuweiri, V.K. Prasanna Kumar, and D. Kim. Orthogonal Multiprocessor Sharing Memory with an Enhanced Mesh for Integrated Image Understanding. *CVGIP: Image Understanding*, 53(1):31–45, January 1991.

[6] K. Hwang and D. DeGroot, editors. *Parallel Processing for Supercomputers and Artificial Intelligence.* McGraw-Hill, New York, March 1989.

[7] K. Hwang, M. Dubois, et al. OMP: A RISC-based Multiprocessor using Orthogonal-Access Memories and Multiple Spanning Buses. In *Proc. of the ACM International Conference on Supercomputing*, pages 7–22, June 1990.

[8] K. Hwang and D. K. Panda. Mapping Multicomputer Communication Patterns onto Multiprocessor as Message Vectors in Shared Memory. *Technical Report* 91-07, Dept. of Electrical Engineering-Systems, Univ. of Southern California, Los Angeles, CA-90089-0781, March 1991.

[9] K. Hwang and S. Shang. Wired-NOR Barrier Synchronization for Designing Large Multiprocessors. *Technical Report* 91-06, Computer Engineering, Dept. of Electrical Engineering-Systems, Univ. of Southern California, Los Angeles, March 1991.

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

[11] Intel Corp. *i860 Development System User's Guide*, 1989.

[12] A. H. Karp. Programming for Parallelism. *IEEE Computer*, 20(5):57, May 1987.

[13] R. H. Katz and J. L. Hennessy. High Performance Microprocessor Architectures. *International Journal of High Speed Electronics*, 1(1):1–17, 1990.

[14] M. Kumar and K. So. Trace Driven Simulation for Studying MIMD Parallel Computers. *Technical Report* CENG 89-31, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, N.Y. 10598, November 1989.

[15] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transaction on Computers*, 28(9):690–691, September 1979.

[16] S. Mehrotra, C. M. Cheng, K. Hwang, M. Dubois, and D. K. Panda. Algorithm-Driven Simulation and Performance Projection of a RISC-based Orthogonal Multiprocessor. In *Proc. of the International Conference on Parallel Processing*, August 1990.

[17] D. K. Panda and K. Hwang. Reconfigurable Vector Register Windows for Fast Matrix Manipulation on the Orthogonal Multiprocessor. In *Proc. of International Conference on Application Specific Array Processors*, Princeton, New Jersey, September 1990.

[18] D. K. Panda and K. Hwang. Fast Data Manipulation in Multiprocessors Using Parallel Pipelined Memories. *Journal of Parallel and Distributed Computing*, 12(2), June 1991.

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

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

[21] C.D. Thompson and H.T. Kung. Sorting on A Mesh-Connected Parallel Computer. *CACM*, 20(4):263–271, 1977.

[22] P.J. Weinberger. Cheap Dynamic Instruction Counting. *AT&T Bell Laboratories Technical Journal*, 63(8):1815–1826, October 1984.

**Kai Hwang** is a Professor of Electrical Engineering and Computer Science at the University of Southern California. Specializing in computer architecture and parallel processing, Professor Hwang has published over 110 papers and 4 books in this area. He serves as the Editor-in-Chief of JPDC in charge of special issues and invited papers. An IEEE Fellow, Dr. Hwang has lectured worldwide and consulted with various national and international research agencies.

**Chien-Ming Cheng** received his BS degree in 1982 from National Taiwan University, Taipei, Taiwan, and his MS degree in Computer Engineering in 1986 from the University of Southern California where he is continuing his Ph.D. research. His major research interests are in parallel processing, parallel programming, and performance evaluation. Mr. Cheng is a student member of both IEEE and ACM.