Mapping Multicomputer Communication Patterns onto Multiprocessors as Message Vectors in Shared Memory[1]

Dhabaleswar K. Panda and Kai Hwang

Technical Report 91-07

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

March 4, 1991

[1]Submitted 10 January 1991 to International Conference on Parallel Processing. This research is supported by National Science Foundation Grant No. MIP89-04172 to the University of Southern California.

Mapping Multicomputer Communication Patterns Onto Multiprocessors as Message Vectors in Shared Memory*

Dhabaleswar K. Panda and Kai Hwang

University of Southern California
Department of EE-Systems
Los Angeles, CA 90089-0781

Abstract: We present a new concept of *communication vectorization*, as a replacement to the scalar-variable approach in a shared-memory multiprocessor. A high-level communication model is developed for multicomputer programs. This model encapsulates all possible communication patterns such as *one-to-one, one-to-all, many-to-many*, and *all-to-all* including *broadcast* or *personalized multicast* options. A mapping methodology is presented to convert *send* and *receive* message-passing operations into equivalent memory *write* and *read* access steps. Multiprocessors with interleaved shared-memory organization can implement these memory access steps as vector operations. Similar to vectorizing computational steps, our method vectorizes message-passing communication steps, which leads to significant reduction in overhead. Three different multiprocessor configurations, *single bus-based, crossbar-connected,* and *orthogonally-connected*, are evaluated with respect to their capabilities to support message vectors in a shared memory. Using this communication vectorization, m-node hypercube and torus are mapped onto an n-processor *orthogonal multiprocessor*, where $m \gg n$. The mapping results in $O(\log m)$ and $O(\sqrt{m})$ reduction in communication cost for hypercube and torus, respectively. The results are useful in converting programs written for multicomputers into multiprocessor programs using the shared memory.

Contents ..... Page

* A preliminary version of this technical report was submitted to the *1991 International Conference on Parallel Processing.* This research is supported by NSF Grant No. MIPS 89-04172 at the University of Southern California from 1989 to 1991.

1. Introduction

It is well known that a multicomputer program can be executed on a shared-memory system by some mapping methodology, where computing nodes of a multicomputer are mapped to processors of the shared-memory system and message-passing operations are carried out by using memory-based mailboxes [22]. However, no work has been done in investigating the effect of architecture parameters of a shared-memory system like *interconnection network, memory access times, data contention*, etc. on the efficiency of such a mapping [2, 5]. Another challenge is whether this mailbox type of *scalar communication* is the best possible solution to the mapping problem at hand.

The crust of the mapping problem depends on efficient conversion of *send* and *receive* operations in a multicomputer program to appropriate *store* and *load* memory operations on a shared-memory system. The mailbox type of scalar communication provides a direct mapping between these two sets of operations. This methodology works well for multicomputer programs involving only *permutation* type of communication. However, most parallel algorithms in scientific and numerical applications involve *broadcast, multicast*, and *personalized multicast* operations [6, 11, 13, 15]. Several routing schemes have been proposed in the literature [4, 14] for efficient implementation of these communication-intensive operations on various multicomputers. The scalar communication, due to network and memory-access conflicts [17], limits the efficiency of mapping *broadcast* and *multicast* communication patterns onto a shared-memory system.

Many shared-memory systems with interleaved memory provide vector-oriented memory accesses, and support vector/matrix data structures at the hardware level. The interleaved memory organization with a smart *on-the-fly index manipulator* [19] have been demonstrated supporting fast data manipulation [20]. The orthogonal multiprocessor [9] with its two-dimensional interleaved memory organization demonstrates capabilities for mapping a large class of multicomputer programs.

In this paper, we take this vector-oriented approach to developing a general framework for mapping message-passing operations of any multicomputer program, irrespective of its underlying architecture, onto various shared-memory multiprocessor systems. We develop a program-level communication model for multicomputers. We define a set of *primitive communication patterns* in a multicomputer environment. The communication steps of a

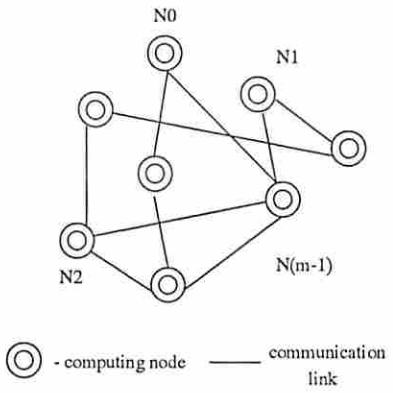


Figure 1: Logical architecture of an n-node multicomputer with an arbitrary message-passing network.

multicomputer program can be reduced using a combination of these primitive patterns. A transformation methodology is provided to convert primitive patterns into message vectors. Using a sample program, we illustrate communication vectorization for three different shared-memory multiprocessor organizations. Finally, we illustrate the mappings of hypercube and torus multicomputer programs into multiprocessors. The efficiency of communication vectorization is determined by analyzing reductions in communication overhead, associated with emulating communication patterns as message vectors.

2. A Communication Model for Multicomputer Programs

Consider a multicomputer system $\mathcal{N}$ consisting of a set of m computer modules and a set of communication links. Each computing node is equipped with its own processing unit, local memory, and hardware for message communication over the message-passing network [1, 7]. Figure 1 shows an arbitrary message-passing network between the nodes of a m-node multicomputer. This network determine the architecture of multicomputer, i.e. hypercube, mesh, torus, and tree, etc.

We consider a partitioned multicomputer program [16, 22] for our mapping purpose. The procedures in the program are assumed to communicate using *send* and *receive* communication constructs. The procedures are assumed to be statically allocated to multicomputer nodes without any dynamic load balancing. We assume the message passing scheme to be *unblocked send* and *blocked receive* [10]. In this scheme, the sender process continues processing immediately on dispatching the message and the receiver process waits for the data to be arrived. The program is also assumed to be deadlock free so that no two processors at

any instant of time wait for messages from each other.

We consider mapping a m-node multicomputer program onto an n-processor shared-memory system, where $m \geq n$. Though our methodology works for any value of m, our illustrations and examples are based on $m = 4$ for easy understanding. Figure 2 shows a sample procedure-level program for a 4-node multicomputer. In each procedure, computation and communication steps alternate. With exchange of messages (through appropriate send and receive constructs), the procedures get synchronized and the program execution proceeds in a wave like manner.

The communication steps in Fig. 2 demonstrate different communication patterns. For an example, the first step is an *one-to-one (personalized)* message exchange between nodes N0 and N1. The second step is a *many-to-many (personalized)* message exchange between two sets of nodes, (N1,N2) and (N3,N0). In the third communication step, node N0 broadcasts a message to rest of the nodes. Node N0 receives personalized messages from all other nodes in the fourth communication step. The last step combines two operations: a *multicast* (broadcasting a message to selected others) operation to nodes N1 and N2 from N3 and a *many-to-one* operation from nodes N1 and N2 to N0.

Depending on the source and destination node sets and the type of communication, i.e. *broadcast* or *multicast*, we can categorize the communication steps of a multicomputer program into various patterns. Table 1 lists all possible communication patterns together with their characteristics identifiers, C1–C12. Use the notation (Nx,Ny) to represent a single message transmission from node Nx to node Ny and ((Nx1,Ny1), (Nx2,Ny2)) to represent transmission of multiple messages from a set of source nodes, Nx1 and Nx2, to a set of destination nodes, Ny1 and Ny2. The communication steps in a multicomputer program can now be represented as a sequence of communication operations. For an example, the multicomputer program in Fig. 2 is represented by following program-level communication operations:

Step 1: $C1(N1, N0)$
Step 2: $C2((N1, N2), (N3, N0))$
Step 3: $C3((N0, N1), (N0, N2), (N0, N3))$
Step 4: $C4((N0, N1), (N0, N2), (N0, N3))$
Step 5: $C10((N1, N0), (N2, N0)), C8((N3, N1), (N3, N2))$

It can be seen that some communication patterns are subsets of others. For an example, *many-to-many* pattern can be expressed as a subset of *all-to-all* with selective de-

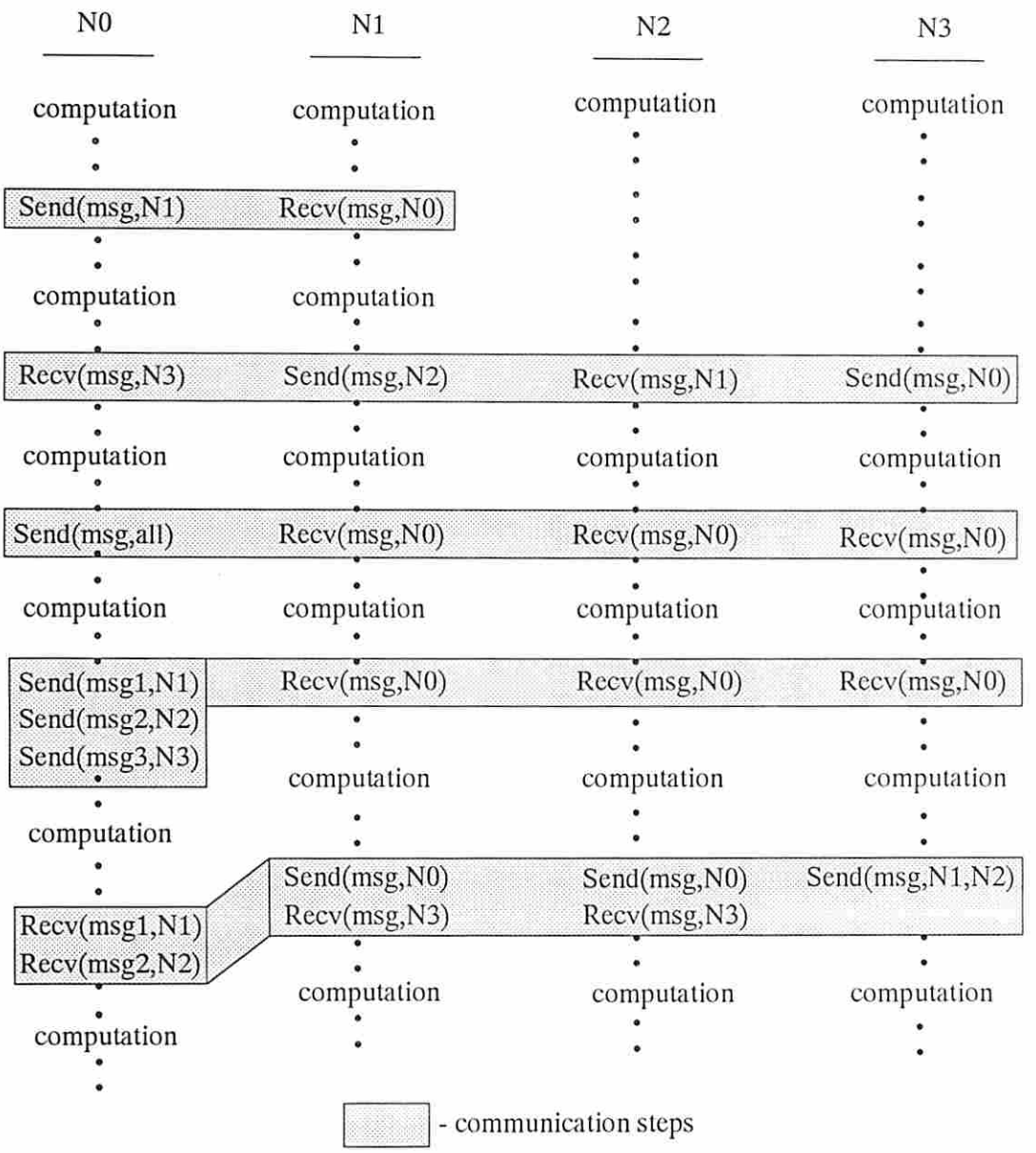


Figure 2: Procedure-level communication steps in a sample program for a 4-node multicomputer (N0-N3: computing nodes).

Table 1: Possible Multicomputer Communication Patterns (m=total number of nodes, sr= number of sources, ds= number of destinations, $1 < sr, ds < m - 1$).

Identifier	Communication pattern	Type	No. of source nodes	No. of destn nodes	No. of distinct messages	Total no. of messages	Degree of replication
C1	one-to-one	personalized	1	1	1	1	1
C2	permutation	personalized	m	m	m	m	1
C3	one-to-all	broadcast	1	m-1	1	m-1	m-1
C4		personalized	1	m-1	m-1	m-1	1
C5	all-to-one	personalized	m-1	1	m-1	m-1	1
C6	all-to-all	broadcast	m	m-1	m	m(m-1)	m-1
C7		personalized	m	m-1	m(m-1)	m(m-1)	1
C8	one-to-many	broadcast	1	ds	1	ds	ds
C9		personalized	1	ds	ds	ds	1
C10	many-to-one	personalized	sr	1	sr	sr	1
C11	many-to-many	broadcast	sr	ds	sr	sr.ds	ds
C12		personalized	sr	ds	sr.ds	sr.ds	1

activation of few nodes. Thus, it is important to identify a set of patterns which are mutually independent and can be used for representing all other patterns. It is obvious that all communication patterns can be represented as multiple steps of *one-to-one* operation. But, we look for a maximal set and denote it as a *minimal Communication Set* (CS). The communication patterns C1–C7 satisfy membership property of the set CS and are defined as *basic communication patterns.* Thus, our program-level communication model of a multicomputer program is identified by the following set:

$$CS = \{C1, C2, C3, C4, C5, C6, C7\}$$

where C1 to C7 are communication patterns characterized in Table 1. All other communication patterns, C8–C12, can be expressed in terms of these basic patterns.

Each of these basic patterns, independent of the underlying multicomputer architecture, can be represented as a *Communication Graph* (CG). Figure 3 shows the communication graphs CG1–CG7. These graphs are unidirectional in nature and represent message passing from a set of source nodes to a set of destination nodes. An appropriate number of switch boxes are used to provide required message transfers. The *indegree* and *outdegree* of each switch box is one. The number of switch boxes used in CGi are equal to total number of

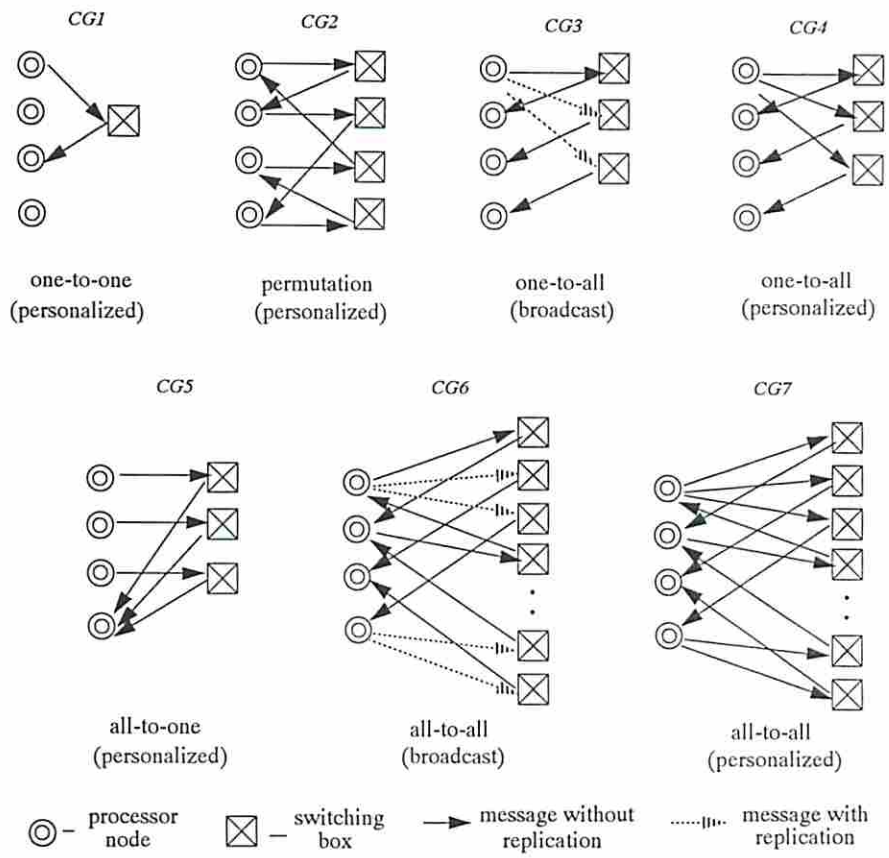


Figure 3: Architecture independent *Communication Graphs* (CGs) for basic communication patterns.

messages involved in communication pattern Ci, $1 \leq i \leq 7$. These communication graphs also demonstrate broadcast or personalized type of message communication.

3. Vectorized Access of Shared Memory

In this section three different multiprocessor configurations using interleaved shared-memories are characterized. Both scalar and vector type of memory accesses are discussed with appropriate cost overheads. A data manipulator to carry out efficient vectorized memory access is emphasized.

3.1 Three Interleaved Memory Organizations

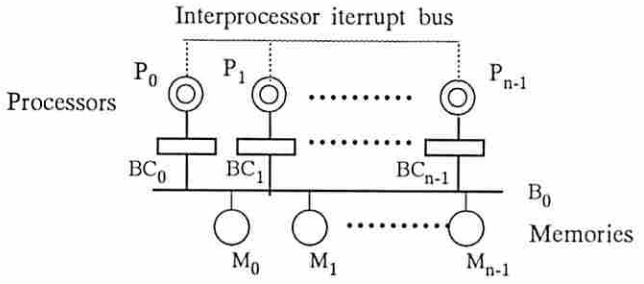
Figure 4 illustrates three shared-memory configurations: single bus-based, crossbar-

connected, and orthogonally-connected. All these configurations use interleaved memories. Consider an n processor system with n memory modules as shown in Fig. 4(a). The n memory modules, $M_0, M_1, \ldots, M_{n-1}$, are connected to a single bus, B_0. The n processors, $P_0, P_1, \ldots, P_{n-1}$, are connected to the bus through n identical *bus controllers*, $BC_0, BC_1, \ldots, BC_{n-1}$. A bus controller supports *interleaved-read* and *write* accesses to n memory modules attached to the bus. Memories are n-way interleaved based on low order S-access scheme. Each processor also has scalar access capability to any of the memory modules. The system provides fully shared-memory capability. An optional interprocessor interrupt bus is used to enable fast synchronization among the processors.

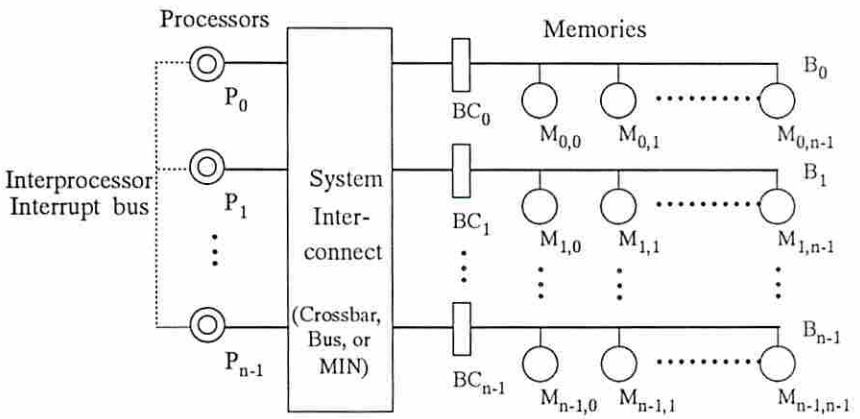
Figure 4(b) represents an n processor system with n^2 memory modules, $M_{0,0}, M_{0,1}, \ldots, M_{n-1,n-1}$. These memory modules are connected to n different buses, $B_o, B_1, \ldots, B_{n-1}$. The n processors, $P_0, P_1, \ldots, P_{n-1}$, are connected to the n buses through a system interconnect. Without any loss of generality, we assume a crossbar-connected interconnect for our analysis. This configuration also provides fully shared-memory capability.

Figure 4(c) shows an orthogonally-connected multiprocessor configuration with n processors, and $2n$ buses. This architecture concept was originally reported in [9]. A design implementation of this orthogonal multiprocessor was reported in [8]. A group of n row buses, $RB_0, RB_1, \ldots, RB_{n-1}$, are directly connected to the processors in the horizontal dimension. The remaining n column buses, $CB_0, CB_1, \ldots, CB_{n-1}$, are distributed across the two-dimensional memory organization in an orthogonal way. Consider indices in the range $0 \leq i, j, k \leq n-1$. A memory module $M_{i,j}$ is connected to two buses, RB_i and CB_j. Both RB_i and CB_i are controlled by the same bus controller BC_i. Both row and column buses support interleaved access to the respective n memory modules connected to them. This configuration allows a memory module $M_{i,j}$ to be shared between two processors P_i and P_j. But, only one of the two processors can access the memory module at a given time. This access scheme makes the multiprocessor a partially shared-memory system.

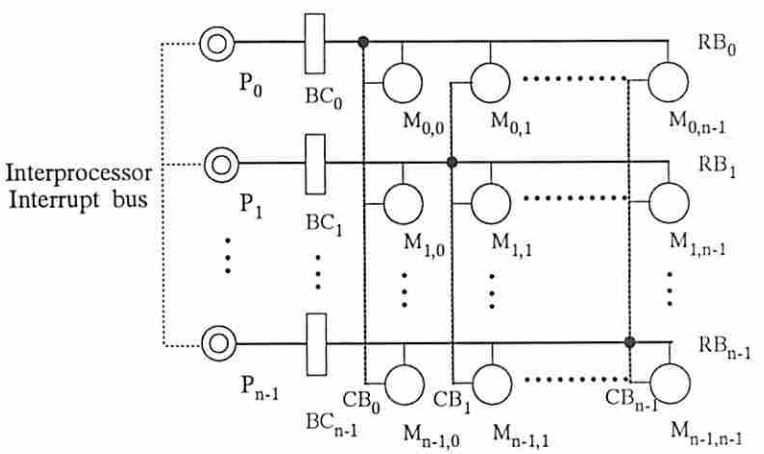
The bus controllers in all three multiprocessor configurations are assumed to support both scalar and vector accesses. Consider a conflict-free scalar memory access by a processor P_i to an interleaved memory module. As shown in Fig. 5(a), there is a fixed time α to initiate this memory access. This time includes time to activate the bus controller, the bus controller putting required address on the bus, address propagation delay on the bus, and memory access time. For a *read* operation, data element from the memory module is read out of the bus with a cycle time of τ. Similarly for a *write* operation, the data is written to



(a) Single bus-based



(b) Crossbar-connected



(c) Orthogonally-connected

Figure 4: Three shared-memory multiprocessor configurations using interleaved memory organizations: (a) Single bus-based multiprocessor, (b) Crossbar-connected multiprocessor, and (c) Orthogonally-connected multiprocessor(P_i: Processor, M_i or $M_{i,j}$: Memory module, B_i: Access Bus, BC_i: Bus Controller).

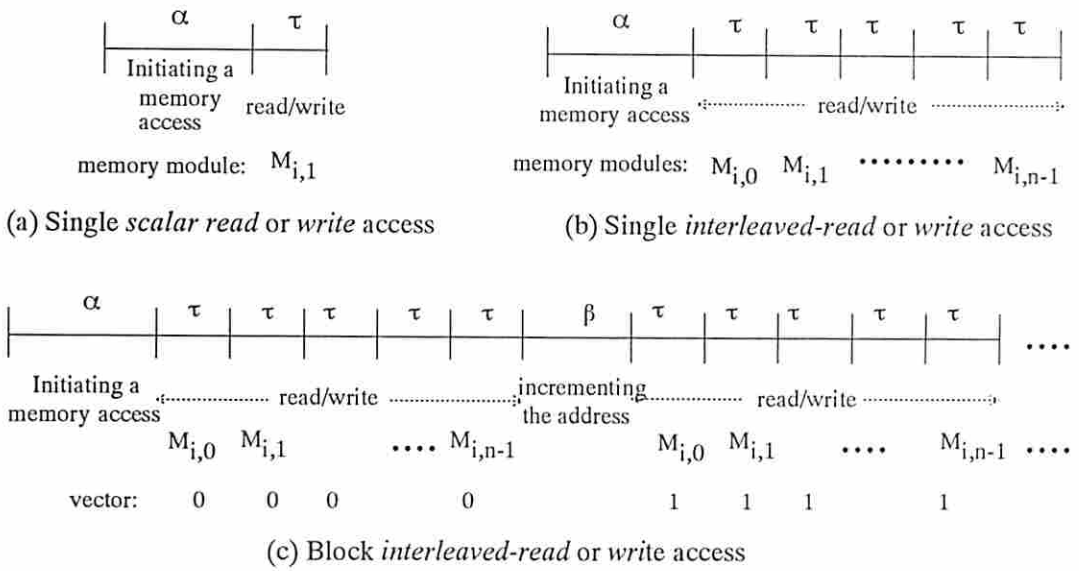


Figure 5: Different protocols for implementing scalar and vector memory access operations.

the bus with a cycle time of τ. Thus, a single scalar memory access takes $(\alpha + \tau)$ time.

Consider an *interleaved-read/write* operation as shown in Fig. 5(b). For a *read* operation, data elements from the memory modules are streamed out of the bus in every minor cycle with a pipelined cycle time of τ. Similarly for a *write* operation, data elements are streamed in via the bus first and then a *parallel-write* operation is activated. This *parallel-write* operation time is included in α as the memory access time. Thus, the overall time to perform a single *interleaved-operation* is $(\alpha + n\tau)$.

Figure 5 (c) shows the protocol for a *block-interleaved-read* operation. In this protocol, a block of vectors are written to consecutive words of the memory modules in consecutive interleaved cycles. We assume an overhead of β to increment subsequent addresses of the memory modules. For a block with l vectors, this block read/write operation takes $(\alpha + \beta(l - 1) + n\tau l)$ time. Using the current bus technology, one can have $\tau = 50$ nsec, $\alpha = 800 - 1000$ nsec and $\beta = 100$ nsec for $n \leq 32$.

Consider this *block-interleaved* memory access operation for implementing message-passing through shared memory. A *block-interleaved-write* operation followed by a *block-interleaved-read* operation allows messages to be exchanged between two sets of processors. We define such a protocol as an *Interleaved-Write-followed by-interleaved-Read* (IWR) step.

The degree of memory interleaving during these interleaved cycles can be up to d memory modules, where $d \leq n$. This allows vector load/store operations to be d bytes long. An IWR step needs a bus switching with additional overhead of $\gamma = 400 - 500$ nsec between two interleaved cycles. Assume $\beta = \delta\tau$. Let $T(d, l)$ represents the time overhead (cost) for a single IWR cycle neglecting the synchronization overhead γ. Assuming $(\alpha - \beta)$ being a *fixed cost* t, we have

$$IWR\ access\ time,\ \ T(d,l) = 2(\alpha + \beta(l-1) + d\tau l) = 2(t + (d+\delta)\tau l) \qquad (1)$$

3.2 Vector Memory Access Using Data Manipulator

Consider pipelined data transfer operations during an *interleaved-read* access. The n elements are loaded from n memory modules to a set of n data buffers associated with the processor. Similarly, during a *store* operation, the data elements are read from different buffers and written to the memory modules. Let e, $0 \leq e \leq n-1$, be the indices of these buffers. Consider any permutation or mapping (ρ) on the index set $E = \{0, 1, \ldots, n-1\}$ with these indices.

As the data elements are transmitted (or received) to (or from) the interleaved bus in a pipelined manner, the source (or destination) buffers can be selected based on ρ. We define ρ as an *index set* and the operation of selecting appropriate buffer during a load/store operation as *indexing*. We provide a novel index manipulator hardware [20], where index sets are generated off-line during compile time as specified by the programmer and stored in fast *index memories* local to the processors. During each memory access, a desired index set is selected from the index memory to implement required buffer manipulation.

Figure 6(a) shows the functional organization of an example index manipulation logic. The interleaved bus is assumed to have 3 memory modules. The processor has 3 data buffers. Figure 6(b) shows the data buffer manipulation scheme during an *interleaved-read* operation with an index set $\{0, 2, 1\}$. As the data elements are read from the pipelined bus in sequence, they are written into the buffers 0,2, and 1 respectively.

Similarly, during an *interleaved write* operation with an index set $\{1, 1, 2\}$, the elements are read from the buffers 1,1, and 2 respectively and written to the memory modules as shown in Fig. 5(c). This feature provides a versatile mechanism for on-the-fly message replication and provides uniform overhead for implementing memory-based broadcast and

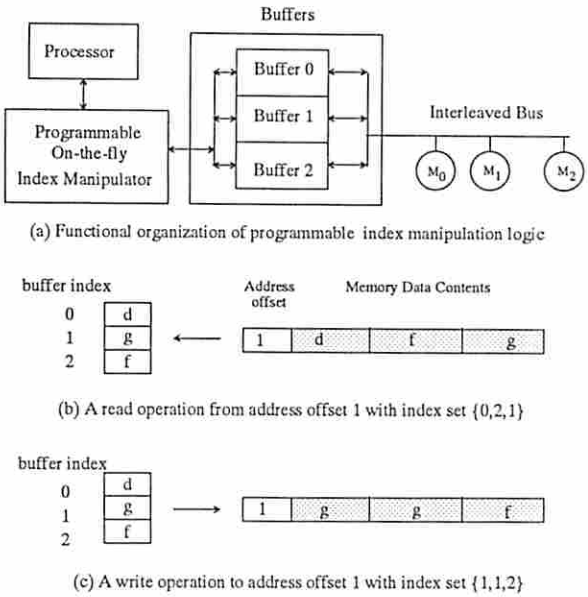


Figure 6: Functional organization and operating principle of an example data buffer manipulator with 3 memory modules on an interleaved bus.

multicast operations. The usage and usefulness of this feature is emphasized in the following section.

4. Mapping of Communication Patterns

In this section, we demonstrate how message passing operations in a multicomputer can be implemented through vectorized shared-memory accesses. We transform the communication graphs presented in Section 2 to independent, *send* and *receive*, memory-based communication graphs. We determine potential capability of implementing these memory-based communication graphs as a single vectorized step in a given multiprocessor. A formal methodology is presented for mapping communication operations onto three shared-memory multiprocessor configurations.

Consider the communication graphs CG1–CG7 in Fig. 3. Each switching box associated with message transmission can be replaced by a shared-memory location. The sender processor writes the message to this memory location and the receiver processor reads it back. These two operations take two time steps of t_s duration each. For single bus-based system, memory module M_i is used for passing messages from P_j to P_i, $1 \leq i, j \leq n$.

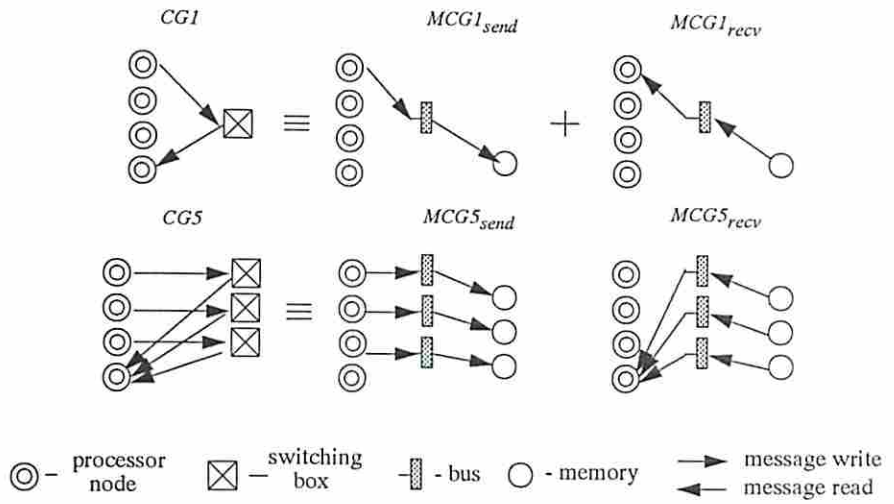


Figure 7: Transformation of communication graphs (CG) to memory-based communication graphs (MCG).

For the other two architectures, memory module $M_{i,j}$ is used for the same purpose. These two *send* and *receive* operations can be represented graphically as *Memory-based Communication Graphs* (MCGs). For a given communication pattern, we transform its CG to MCG. For each basic communication pattern, Ci, in the communication set CS, we represent its equivalent scalar memory-based communication as $MCGi_{send}$ and $MCGi_{recv}$ graphs, where $1 \leq i \leq 7$. Figure 7 illustrates the graphs for communication patterns C1 and C5.

4.1 Vectorized Communication via Shared Memory

In section 3.1, we discussed about interleaved (vector) read/write accesses in shared-memory configurations. An on-the-fly data manipulator was also presented in section 3.2. Here we show the potential of this data manipulator to support vectorized memory-based communications. Consider the memory-based communication graph $MCG5_{recv}$ as shown in Fig. 7. If all memory modules are connected to a single bus, the destination processor can perform an *interleaved read* operation to all memory modules and receive message from all others.

In the absence of vectorized memory access, this step is performed by many (3 for this example) scalar memory accesses. Thus, vectorized memory access provides efficiency in implementing communication-intensive patterns (C3–C7) on shared-memory multiproces-

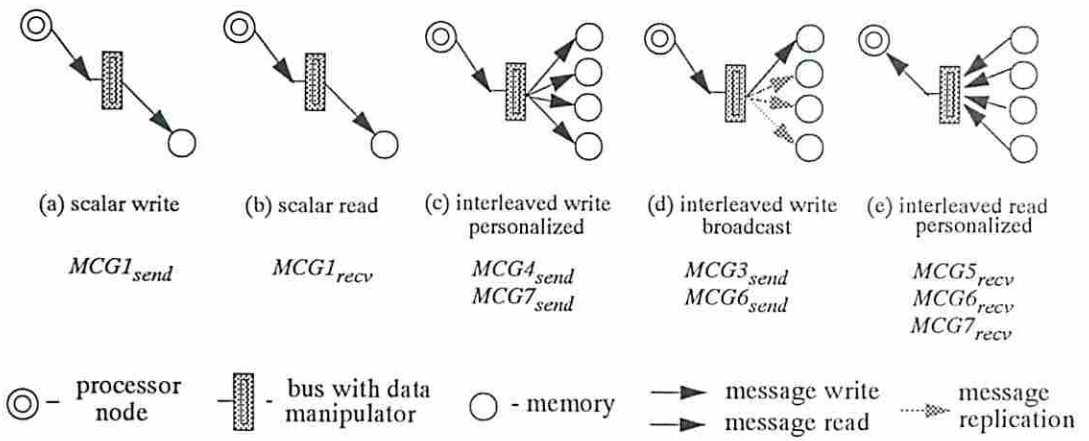


Figure 8: Different functionality of vectorized memory access with on-the-fly data manipulator and the potential for implementing equivalent memory-based communication graphs (MCGs) in single vectorized memory access step.

sors. Figure 8 shows five different functionality of vectorized memory access with the data manipulator hardware. Functional equivalent memory-based communication graphs are also shown. Each of these functional equivalent communication operations, corresponding to these graphs, demonstrate potential for being implemented in a single vectorized memory access step.

Whether a given communication pattern can be vectorized or not on a multiprocessor depends on the interleaved memory organization and the processor-memory interconnection network of the multiprocessor. We denote this vectorization capability as *vectorizability*. For a given multiprocessor architecture, this vectorizability can be determined as follows: First, we generate a *structural connectivity graph* for a multiprocessor representing its connectivity between processors, buses, and memory modules. In the second step, all possible *operational subgraphs* of this structural connectivity graph, satisfying the hardware operational constraints for vector memory access, are generated. Finally, the memory-based communication graph corresponding to a communication pattern is checked against all operational subgraphs for a match. In case of a match, the associated communication step demonstrates potential for vectorization. Otherwise, this step is implemented as scalar memory communication. In the following section, we analyze vectorizability for primitive communication patterns for three multiprocessor configurations. A detailed graph-theoretic approach to this vectorizability scheme for general multiprocessor organizations is under preparation as

a follow-up paper.

4.2 The Mapping Methodology

Consider an n-processor target shared-memory multiprocessor X. An n-node multicomputer program is mapped to X by an one-to-one mapping of the computational steps of multicomputer nodes to processors in X. We are interested in mapping multicomputer communication operations through shared-memory message passing. As we discussed in section 2.3, communication operations of a multicomputer program can always be expressed in terms of the basic communication patterns C1–C7 in the communication set CS. In the last subsection, we emphasized on memory-based communication graphs for these basic patterns. We also discussed that some of these memory-based communication graphs demonstrate potential for being implemented in single vectorized memory access step. These vector access steps are efficient (with less time overhead) compared to equivalent multiple scalar access steps.

The objective of an optimal mapping using vectorized memory-communication is to reduce overall communication time. This demands that maximum number of communication operations should be implemented in vectorized manner. It is also clear that transformation of multicomputer communication patterns into vectorized memory communication depends on the architecture X and the communication pattern itself. We propose the following 4-step methodology to map a multicomputer program onto a target shared-memory architecture X.

1. Reduction of communication operations of a multicomputer program in terms of basic communication patterns in the communication set CS.
2. Determining a subset of these basic patterns that can be implemented on X using only scalar access. Determining another subset which demonstrate potential for being implemented using vector access.
3. Transforming communication operations of a program into equivalent memory-based communication steps by using optimal combination of scalar and vector accesses.
4. Restructuring memory communication steps with appropriate synchronization primitives.

First we discuss step 2 of this methodology which is most crucial. In the next subsection, we illustrate the complete methodology by mapping the sample program, presented in section 2, onto three multiprocessor configurations.

Consider the communication patterns C1–C7 in the communication set CS. The *send* and *receive* operation, associated with each communication pattern, gets transformed into equivalent *write* and *read* steps in a shared-memory environment. We identify *send* and *receive* steps of each communication pattern as Ci_{send} and Ci_{recv}, where $1 \leq i \leq 7$. The equivalent transformations into memory-based communication are represented in terms of memory-based communication graphs as shown in Fig. 7.

Similar to the concept of communication set CS, we define a *Memory-based Communication Set* (MCS). For a given shared-memory architecture X, we define two such sets, MCS_X^{scalar} and MCS_X^{vector}, for scalar and vector operations respectively. MCS_X^{scalar} comprises of all *send* and *receive* operations of basic patterns that can be implemented on the architecture X using single-step scalar access only. Similarly, MCS_X^{vector} encapsulates the respective *send* and *receive* operations that can be implemented using single-step vectorized access. The following six memory-based communication sets comprise basic send and receive communication patterns that can be implemented on bus-based, crossbar-connected, and orthogonally-connected multiprocessor systems, using a single-step scalar/vector memory-access.

1. $MCS_{bus}^{scalar} = \{C1_{send}, C1_{recv}\}$
2. $MCS_{bus}^{vector} = \{C3_{send}, C4_{send}\}$
3. $MCS_{crossbar}^{scalar} = \{C1_{send}, C1_{recv}, C2_{send}, C2_{recv}\}$
4. $MCS_{crossbar}^{vector} = \{C3_{send}, C4_{send}, C6_{send}, C7_{send}\}$
5. $MCS_{orthogonal}^{scalar} = \{C1_{send}, C1_{recv}, C2_{send}, C2_{recv}, C3_{recv}, C4_{recv}, C5_{send}\}$
6. $MCS_{orthogonal}^{vector} = \{C3_{send}, C4_{send}, C5_{recv}, C6_{send}, C6_{recv}, C7_{send}, C7_{recv}\}$

Consider the basic patterns, $Ci, 1 \leq i \leq 7$. Memory-based message communication belonging to *send* and *receive* operations of these communication patterns can be represented by unidirectional memory-based communication graphs, $MCGi_{send}$ or $MCGi_{recv}$, as shown in Fig. 7. We assume the use of data manipulator hardware in all three multiprocessor configurations to support vectorized memory access. Out of 14 (7 pairs of *send* and *receive*) possible communication operations, we separate out those operations which demonstrate vectorization capability. This vector group includes 7 operations: $MCG3_{send}, MCG4_{send}, MCG5_{recv}$,

$MCG6_{send}, MCG6_{recv}, MCG7_{send}$, and $MCG7_{recv}$. The remaining operations are included in the scalar group.

4.3 Mapping into Message Vectors

The concept and properties of a message vector are explained as follows: Let $\mathcal{N} = \{N0, N1, \ldots, N(m-1)\}$ be the set nodes in a m-node multicomputer. The set $\mathcal{E} = \{(N0 - N1), (N0 - N2), \ldots, (N(m-1) - N(m-2)\}$ represents all *point-to-point* communication links. The power set $2^{\mathcal{E}} = \{\phi, \{(N0 - N1)\}, \{(N0 - N2)\}, \ldots, \{(N0 - N1), (N0 - N2)\}, \ldots, \{(N0 - N1), (N0 - N2), \ldots, (N(m-1) - N(m-2)\}\}$ represents all possible *communication patterns* using different combination of the links in the set $\mathcal{E}$. A message vector v is a collection of one or more communication patterns, which can be implemented in one vectorized step as discussed in Section 4.3. The pattern v can not be null. The maximum number of links used in a given v is $m(m-1)$. Let DST_i be the set of all *destination* nodes from the source node i. Similarly, let SRC_i be the set of all *source* nodes destining for node i.

Theorem 1 *A message vector, v, must satisfy the following four properties:*

1. $(Ni, Nj1), (Ni, Nj2) \in v, \ \forall i, 0 \leq i \leq m-1 \ \text{ iff } \ j1 \neq j2$
2. $(Ni1, Nj), (Ni2, Nj) \in v, \ \forall j, 0 \leq j \leq m-1 \ \text{ iff } \ i1 \neq i2$
3. $DST_i = \{j \mid (Ni, Nj) \in v, \ 0 \leq j \leq m-1\}, \forall i, \ 0 \leq i \leq m-1 \ \text{ iff } \ |DST_i| \leq m-1$
4. $SRC_j = \{i \mid (Ni, Nj) \in v, \ 0 \leq i \leq m-1\}, \forall j, \ 0 \leq j \leq m-1 \ \text{ iff } \ |SRC_j| \leq m-1$

Proof:

Necessity: During one vectorized access, a processor P_i can either write a single message to the memory module $M_{i,j}$ or receive a single message from the memory module $M_{i,j}$. Hence, for any source (destination), all destinations (sources) have to be disjoint as indicated by properties 1 and 2. During one vectorized access on an m-way interleaved bus, any source can send messages to at most $(m-1)$ other destinations as indicated by property 3. Similar arguments hold good for receiving nodes indicated by property 4.

Sufficiency: The vector v satisfying the above four properties guarantees that there exists one distinct memory module in the shared-memory multiprocessor, which can always be allocated to work as a unique buffer for each point-to-point communication belonging to

v. All actively sending nodes can initiate simultaneous *interleaved-write* operation to the designated memory modules in their respective rows, if possible. Similarly, all actively receiving nodes perform *interleaved-read* operation in parallel from the designated memory modules in their respective columns. These two interleaved operations constitute an IWR step. ■

The messages corresponding to the communication operations belonging to memory-based communication sets MCS_{bus}^{vector}, $MCS_{crossbar}^{vector}$, and $MCS_{orthogonal}^{vector}$ constitute message vectors. Based on these message vectors, we take the sample program presented in Fig. 2 to demonstrate the complete mapping methodology.

Consider the program-level communication steps of the sample program as described in Section 2.2. Communication patterns C8 and C10 are not basic patterns. Hence, these patterns are first decomposed/encapsulated using basic patterns. Communication pattern C8, *one-to-many (personalized)*, is encapsulated into C3, which represents *one-to-all (personalized)* communication. Similarly, C10 is encapsulated into C5. So our sample program involves basic communication patterns C1, C2, C3, C4, and C5. Next we consider mapping this communication model, with 5 basic communication patterns, into three different multiprocessor configurations.

A. Single Bus Multiprocessor System:

For a single bus-based system, $C1_{send}$ and $C1_{recv}$ operations can only be implemented in scalar mode. Similarly, $C3_{send}$ and $C4_{send}$ operations demonstrate potential for vector access. That leaves all other operations, $C2_{send}, C2_{recv}, C3_{recv}, C4_{recv}$, $C5_{send}$, and $C5_{recv}$, to be implemented as multiple $C1$ operations. Based on these constraints, communication operations of the sample program are mapped to a single bus-based system as follows [2]:

Step 1: $C1_{send}(P1 \rightarrow M_0)$, $C1_{recv}(M_0 \rightarrow P0)$
Step 2: $C1_{send}(P1 \rightarrow M_2)$, $C1_{send}(P3 \rightarrow M_0)$,
$C1_{recv}(M_2 \rightarrow P2)$, $C1_{recv}(M_0 \rightarrow P0)$
Step 3: $C3_{send}(P0 \rightarrow M_1, M_2, M_3)$, $C1_{recv}(M_1 \rightarrow P1)$,
$C1_{recv}(M_2 \rightarrow P2)$, $C1_{recv}(M_3 \rightarrow P3)$
Step 4: $C4_{send}(P0 \rightarrow M_1, M_2, M_3)$, $C1_{recv}(M_1 \rightarrow P1)$,
$C1_{recv}(M_2 \rightarrow P2)$, $C1_{recv}(M_3 \rightarrow P3)$
Step 5: $C1_{send}(P1 \rightarrow M_0)$, $C1_{send}(P2 \rightarrow M_0)$, $C1_{recv}(M_0 \rightarrow P0)$, $C1_{recv}(M_0 \rightarrow P0)$,

[2]We use the following notations in our memory-based message communication: $(Px \rightarrow M_y)$ for *scalar write* operation from processor Px to memory module M_y, $(Px \rightarrow M_y, M_z, \ldots, M_w)$ for *vector write* operation, $(M_y \rightarrow Px)$ for *scalar read* operation by processor Px from memory module M_y, and $(M_y, M_z, \ldots, M_w \rightarrow Px)$ for *vector read* operation.

$C3_{send}(P3 \rightarrow M_1, M_2)$, $C1_{recv}(M_1 \rightarrow P1)$, $C1_{recv}(M_2 \rightarrow P2)$

B. Crossbar-connected Multiprocessor System:

Using the above described procedure, communication operations of the sample program is mapped to a crossbar-connected system as follows:

Step 1: $C1_{send}(P1 \rightarrow M_{1,0})$, $C1_{recv}(M_{1,0} \rightarrow P0)$
Step 2: $C2_{send}((P1 \rightarrow M_{1,2}), (P3 \rightarrow M_{3,0}))$,
$C2_{recv}((M_{1,2} \rightarrow P2), (M_{3,0} \rightarrow P0)$
Step 3: $C3_{send}(P0 \rightarrow M_{0,1}, M_{0,2}, M_{0,3})$, $C1_{recv}(M_{0,1} \rightarrow P1)$,
$C1_{recv}(M_{0,2} \rightarrow P2)$, $C1_{recv}(M_{0,3} \rightarrow P3)$
Step 4: $C4_{send}(P0 \rightarrow M_{0,1}, M_{0,2}, M_{0,3})$, $C1_{recv}(M_{0,1} \rightarrow P1)$,
$C1_{recv}(M_{0,2} \rightarrow P2)$, $C1_{recv}(M_{0,3} \rightarrow P3)$
Step 5: $C1_{send}(P1 \rightarrow M_{1,0})$, $C1_{send}(P2 \rightarrow M_{2,0})$, $C1_{recv}(M_{1,0} \rightarrow P0)$, $C1_{recv}(M_{2,0} \rightarrow P0)$,
$C3_{send}(P3 \rightarrow M_{3,1}, M_{3,2})$, $C1_{recv}(M_{3,1} \rightarrow P1)$, $C1_{recv}(M_{3,2} \rightarrow P2)$

C. Orthogonal Multiprocessor System:

This shared-memory configuration, with two-dimensional memory interleaving, includes all basic patterns into the sets MCSs. This allows a straight forward mapping of multicomputer message passing operations into memory-based communication. Communication operations of the sample program is mapped to this configuration as follows:

Step 1: $C1_{send}(P1 \rightarrow M_{1,0})$, $C1_{recv}(M_{1,0} \rightarrow P0)$
Step 2: $C2_{send}((P1 \rightarrow M_{1,2}), (P3 \rightarrow M_{3,0}))$,
$C2_{recv}((M_{1,2} \rightarrow P2), (M_{3,0} \rightarrow P0)$
Step 3: $C3_{send}(P0 \rightarrow M_{0,1}, M_{0,2}, M_{0,3})$, $C3_{recv}((M_{0,1} \rightarrow P1), (M_{0,2} \rightarrow P2), (M_{0,3} \rightarrow P3))$,
Step 4: $C4_{send}(P0 \rightarrow M_{0,1}, M_{0,2}, M_{0,3})$, $C4_{recv}((M_{0,1} \rightarrow P1), (M_{0,2} \rightarrow P2), (M_{0,3} \rightarrow P3))$,
Step 5: $C5_{send}((P1 \rightarrow M_{1,0}), (P2 \rightarrow M_{2,0}))$, $C5_{recv}(M_{1,0}, M_{2,0} \rightarrow P0)$
$C3_{send}(P3 \rightarrow M_{3,1}, M_{3,2})$, $C3_{recv}((M_{3,1} \rightarrow P1), (M_{3,2} \rightarrow P2)$

In all the above mappings, we did not show explicit synchronization constructs for clarity. Since each (scalar and vector) shared-memory communication gets implemented in a producer-consumer manner, we need explicit synchronization between them. As shown in Fig. 4, we assume the existence of an interprocessor interrupt bus in all three configurations. We assume a static barrier MIMD hardware synchronization scheme as proposed in [18].

Each receiver processor executes a *wait-sync* operation on the sender processor before a *message read* operation. Similarly, every sender processor executes an *activate-sync* operation after each *message write* operation. Since we assume our original multicomputer

program to be deadlock-free, this synchronization scheme ensures no deadlock in a shared-memory environment. These synchronization primitives are incorporated into the code appropriately before execution.

5. Mapping Multicomputer Programs into Multiprocessors

Based on the communication vectorization concept, we demonstrate mapping different multicomputer architectures to the Orthogonal multiprocessor. Consider the target multicomputer architecture $\mathcal{G}$ with m computer nodes being mapped onto the host OMP architecture $\mathcal{H}$ with n processors, where $m \geq n$. We use a two-step approach. In the first step, the m computer nodes are grouped into n clusters with (m/n) nodes assigned to each cluster. The second step involves allocating these clusters to the processors in $\mathcal{H}$. This approach is similar to the classical problem of mapping tasks to the nodes in a distributed computing environment. Our approach is distinguished from the old methods in that we emphasize on interprocessor vector communication. We determine the efficiency of communication vectorization by emulating frequently used communication patterns as summarized in Table 1. We compare our cost with near-optimal algorithms available in the literature [21]. Asymptotic reduction in communication overhead for short and long messages are derived.

Consider the messages on the target architecture $\mathcal{G}$ to have uniform length l. If $m > n$, multiple computer nodes of the target multicomputer get mapped to a single processor on the host. This exhibits potential increase in the effective message length l by a factor λ. This λ can be defined as a *node-congestion* factor and its value depends on the communication pattern. So, the cost of implementing an IWR access on the OMP changes to $T(d, \lambda l)$. Table 2 shows the cost $T(d, \lambda l)$ to emulate different communication patterns on an OMP under the communication vectorization, where T was defined in Eq. 1.

5.1 Mapping Hypercube Programs onto The OMP

Consider a p-dimensional hypercube with $m = 2^p$ nodes as a target multicomputer $\mathcal{G}$. Let the p dimensions be named as $x0, x1, \ldots, x(p-1)$. We consider two cases of hypercube mapping.

Case A: $(m = n)$

The hypercube consists of n nodes and either $n \log n$ unidirectional links or $(n \log n)/2$ bidirectional links. Each hypercube node (N) is mapped to an orthogonal processor (P). Figure 9 demonstrates such mapping on a 16-processor OMP. The four dimensions $(x0, x1, x2,$

Table 2: Cost of Emulating Communication Patterns Under Communication Vectorization Scheme (m= number of nodes on the target architecture, n = number of processors on the host OMP, l = uniform message length on the target architecture, T is the time overhead for an IWR access as defined in Equation 1).

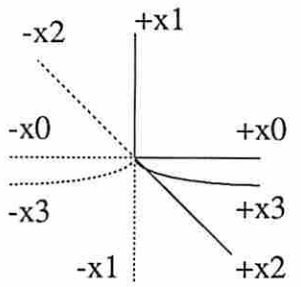
Communication Patterns	Cost	
	$m = n$	$m > n$
One-to-one	$T(1,l)$	0, if source and destination nodes are in the same cluster
		$T(1,l)$
One-to-all (broadcast)	$T(n-1,l)$	$T(n-1,l)$
All-to-all (broadcast)	$T(n-1,l)$	$T(n-1,\frac{m}{n}l)$
One-to-all (personalized)	$T(n-1,l)$	$T(n-1,\frac{m}{n}l)$
All-to-all (personalized)	$T(n-1,l)$	$T(n-1,(\frac{m}{n})^2 l)$

and $x3$) are numbered positively and negatively with respect to the link directions of node 0. This numbering scheme identifies the mapping of unidirectional links to the orthogonal memory modules. For example, the memory module $M_{1,9}$ is used for N_1 to communicate with N_9 using the $+x3$th dimension. The memory module $M_{9,1}$ replaces the -$x3$th dimension link from N_9 to N_1.

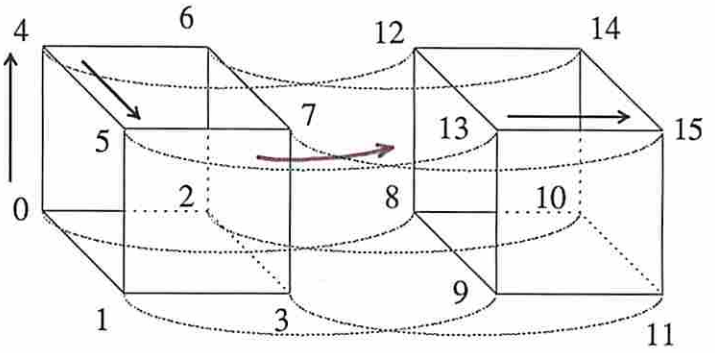
This mapping scheme provides bidirectional multi-port ($\log n$ ports per node) communication. In one IWR step, all n nodes can simultaneously send and receive messages to and from all its $\log n$ neighbors. The communication vectorization supports the hypercube $\log n$-port communication model [21] completely. In fact, the capability of this mapping scheme is much more powerful. Consider a message communication between node 0 and node 15 on a 16-node hypercube in Fig. 9. The communication needs 4 steps on a hypercube. But, the communication vectorization provides a virtual link between node 0 to node 15 through the memory module $M_{0,15}$. Hence, a 4 step communication on a target hypercube can be replaced with a single IWR step under this mapping scheme. In fact, for $m = n$, virtual links exist from each node to every other node. Unlike the *edge-congestion* factor in other mapping schemes, the current mapping provides a *contraction* factor of $\log n$ and allows significant reduction in communication cost in implementing dense communication patterns.

Case B: ($m > n$)

Consider a p dimensional hypercube, $m = 2^p$, to be mapped onto a n processor, $n = 2^q$, OMP. According to the mapping procedure mentioned earlier, the first step groups each 2^{p-q} hypercube nodes into a cluster. The second step allocates these 2^q clusters to n



link dimensions



A 16-node (N) hypercube

P	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		+x2	+x0		+x1				+x3							*
1	-x2			+x0		+x1				+x3						
2	-x0			+x2			+x1				+x3					
3		-x0	-x2					+x1				+x3				
4	-x1					+x2	+x0						+x3			
5		-x1			-x2			+x0						+x3		
6			-x1		-x0			+x2							+x3	
7				-x1		-x0*	-x2									+x3
8	-x3									+x2	+x0		+x1			
9		-x3							-x2			+x0		+x1		
10			-x3						-x0			+x2			+x1	
11				-x3						-x0	-x2					+x1
12					-x3				-x1					+x2	+x0	
13						-x3				-x1			-x2			+x0
14							-x3				-x1		-x0			+x2
15								-x3				-x1		-x0	-x2	

– An *interleaved-write-read* (IWR) step replacing existing hypercube link

— An IWR step providing virtual link

Figure 9: Mapping of a 4-dimensional hypercube onto a 16-processor OMP.

orthogonal processors. This implies that hypercube communication corresponding to $(p-q)$ dimensions are collapsed into intra-cluster communication. The communication corresponding to the rest q dimensions are mapped to the memory modules of the OMP. The first clustering step has $C^p_{(p-q)}$ options to choose the $(p-q)$ dimensions. Figure 10 indicates the mapping of a 32-node hypercube onto a 16-processor OMP. Five options for collapsing any one of the five dimensions to intra-cluster communication are demonstrated. The dimensions belonging to inter-cluster communication are mapped onto the memory modules similar to Fig. 9. The frequently used $(p-q)$ dimensions of a hypercube can be reduced to intra-cluster communication to provide a significant reduction in communication overhead.

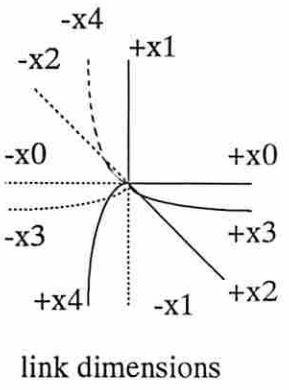
5.2. Reduction in Communication Overhead

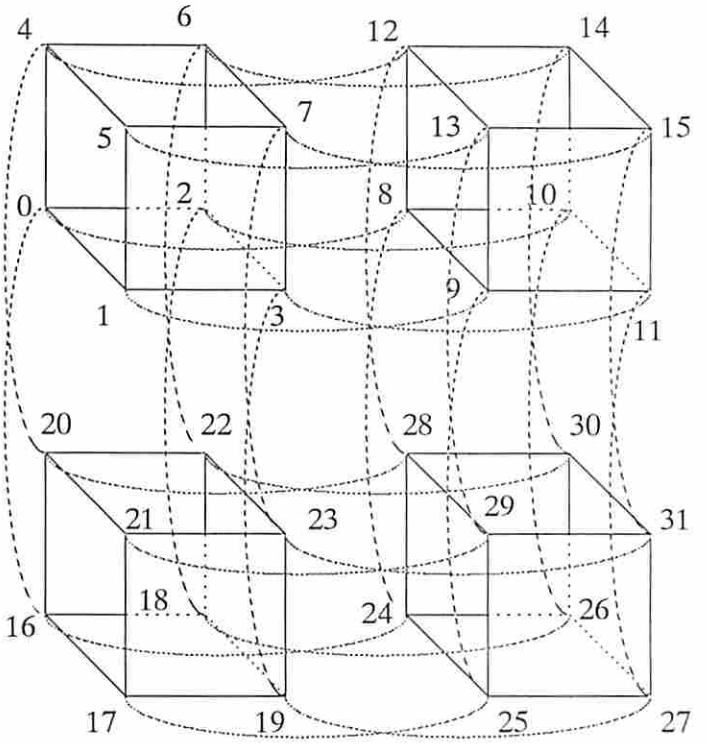
The cost expression T under communication vectorization, as defined in Eq. 1, uses two cost parameters: fixed cost t and pipelined cost τ. We use a similar convention scheme to reflect the message initiation cost in hypercube as t_h and the pipelined communication cost in hypercube links as τ_h. We normalize these cost parameters using constants k_1, k_2, and k_3 as defined below:

$$t = k_3\tau, \quad \tau_h = k_2\tau, \quad t_h = k_1 t = k_1 k_3 \tau \tag{2}$$

where the constants k_1, k_2, and k_3 depend on the design parameters used. Using the above normalization, we express both costs (the cost of emulating communication patterns under communication vectorization and the cost derived by Saad and Schultz in [21]) in terms of τ. The results in [21] assume that for each communication pattern, the total amount of data being exchanged between all processors remains constant. But in our analysis, we assume that each node in $\mathcal{G}$ exchanges data in message of length l irrespective of the communication pattern. So, we normalize the message length appropriately in the results in [21] For example, the total amount of data exchanged in an *all-to-all (broadcast)* communication is lm, in an *one-to-all (personalized)* communication (scatter) is lm, and in an *all-to-all (personalized)* communication (multiscatter) is lm^2.

The multiport hypercube communication models in [21] assumes up to $\log m$ message communication during a single pipelined cycle of τ_h. These models also assume packet disassembly at the source node and packet assembly at intermediate and destination nodes. Up to $\log m$ packets are involved in this assembly and disassembly process. Contrary to this communication model, the pipelined cost τ under communication vectorization involves





A 32-node (N) hypercube

Different Mapping Options

Options	Intra-cluster	Inter-cluster
1	+x0, -x0	+x1, -x1, +x2, -x2, +x3, -x3, +x4, -x4
2	+x1, -x1	+x0, -x0, +x2, -x2, +x3, -x3, +x4, -x4
3	+x2, -x2	+x0, -x0, +x1, -x1, +x3, -x3, +x4, -x4
4	+x3, -x3	+x0, -x0, +x1, -x1, +x2, -x2, +x4, -x4
5	+x4, -x4	+x0, -x0, +x1, -x1, +x2, -x2, +x3, -x3

Figure 10: Mapping of a 5-dimensional hypercube onto a 16-processor OMP.

sending (receiving) a single word to (from) the memory modules. Hence for a fair comparison between these two communication models, we assume $k_2 = O(\log m)$. Without loss of generality, this approximation is useful in comparing the communication costs for long messages. Table 3 analyzes the cost comparison between hypercube communication and the memory-based emulated communication.

Theorem 2 *The mapping of a m-node hypercube multicomputer onto an n-processor OMP results in $O(\log m)$ reduction in communication cost with respect to each of the communication patterns listed in Table 3.*

Proof: Table 3 analyzes the cost comparison between hypercube communication and the memory-based emulated communication. The cost factors shown in the table are normalized to τ_o units. Each cost term involves a fixed cost which is independent of the message length l. The variable cost depends on l. For small message lengths, the fixed costs dominate in both actual and emulated communication. The ratio of the fixed cost of actual hypercube communication to the fixed cost of memory-based emulation determines the cost reduction factor. For long messages, the respective variable costs dominate and determine the cost reduction. For short messages, the comparison demonstrates $O(\log m)$ reduction in communication cost. ■

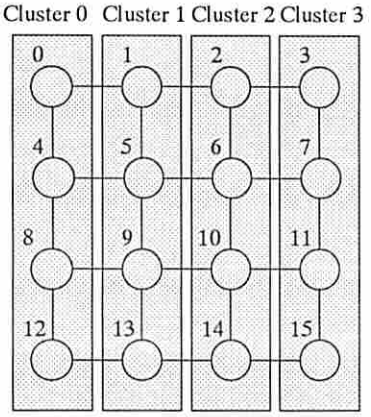
5.3 Mapping of Torus Programs onto The OMP

For $m = n$, the mesh/torus mapping scheme is similar to that of the hypercube mapping shown in Fig. 9. The only difference being the allocation of memory modules based on *east,west,north*, and *south* dimensions. Hence, we directly show the mapping of mesh for $m = n^2$. We consider an $(n \times n)$ mesh with wrap-around torus interconnections. The four communication (rotate) steps are identified as *east, west, north*, and *south*. The n^2 nodes are first grouped into n clusters either by rows or by columns. These clusters are allocated to n processors of an OMP. Figure 11(a) shows an example of clustering by columns. The 16 nodes are grouped into 4 clusters and allocated to 4 processors of the OMP.

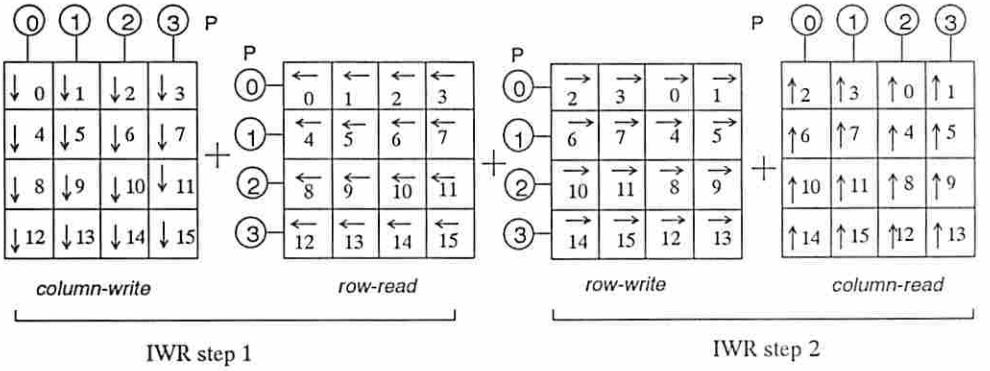
If nodes belonging to a column (or row) are grouped into a single cluster, all intra-column (or intra-row) communication reduces to intra-cluster communication on the OMP. The other communication patterns are implemented as inter-cluster communication. With the example shown, all *north* and *south* communication are reduced to intra-cluster communication. The *east* and *west* communication steps are implemented as inter-cluster commu-

Table 4: Cost Comparison and the associated Speedup between Hypercube Communication and Emulated Communication on the OMP Based on Message Vectors (m= number of nodes on target hypercube architecture, n = number of processors on the host OMP architecture, l = uniform message length on target architecture, k_1, k_2, k_3 = normalizing constants between the fixed and pipelined cost between two schemes, τ = pipelined memory access cost in OMP).

Communication patterns	Hypercube communication cost [21] in τ units	Vectorized communication cost in τ units	Communication cost reduction(short message)	Communication cost reduction(long message) $k_2 = O(\log m)$
$m > n$				
One-to-one	$(2+\log m)k_1k_3 + \frac{2l}{\log m}k_2$	$2k_3 + 2(1+\delta)l$	$\simeq ((2+\log m)/2)k_1 = O(\log m)$	$\simeq \frac{k_2}{\log m(1+\delta)} = O(1)$
One-to-all (broadcast)	$2(\log m)k_1k_3 + 2lk_2$	$2k_3 + 2(n-1+\delta)l$	$\simeq (\log m)k_1 = O(\log m)$	$\simeq \frac{k_2}{n-1+\delta} = O(\frac{\log m}{n})$
All-to-all (broadcast)	$2(\log m)k_1k_3 + 2l\frac{m}{\log m}k_2$	$2k_3 + 2(n-1+\delta)\frac{m}{n}l$	$\simeq (\log m)k_1 = O(\log m)$	$\simeq \frac{nk_2}{\log m(n-1+\delta)} = O(1)$
One-to-all (personalized)	$(\log m)k_1k_3 + lmk_2$	$2k_3 + 2(n-1+\delta)\frac{m}{n}l$	$\simeq ((\log m)/2)k_1 = O(\log m)$	$\simeq \frac{k_2n}{2(n-1+\delta)} = O(\log m)$
All-to-all (personalized)	$2(\log m)k_1k_3 + lmk_2$	$2k_3 + 2(n-1+\delta)(\frac{m}{n})^2l$	$\simeq (\log m)k_1 = O(\log m)$	$\simeq \frac{k_2n^2}{2(n-1+\delta)m} = O(\frac{n\log m}{m})$
$m = n$				
One-to-one	$(2+\log n)k_1k_3 + \frac{2l}{\log n}k_2$	$2k_3 + 2(1+\delta)l$	$\simeq ((2+\log n)/2)k_1 = O(\log n)$	$\simeq \frac{k_2}{(\log n)(1+\delta)} = O(1)$
One-to-all (broadcast)	$2(\log n)k_1k_3 + 2lk_2$	$2k_3 + 2(n-1+\delta)l$	$\simeq (\log n)k_1 = O(\log n)$	$\simeq \frac{k_2}{n-1+\delta} = O(\frac{\log n}{n})$
All-to-all (broadcast)	$2(\log n)k_1k_3 + 2l\frac{n}{\log n}k_2$	$2k_3 + 2(n-1+\delta)l$	$\simeq (\log n)k_1 = O(\log n)$	$\simeq \frac{nk_2}{(\log n)(n-1+\delta)} = O(1)$
One-to-all (personalized)	$(\log n)k_1k_3 + lnk_2$	$2k_3 + 2(n-1+\delta)l$	$\simeq ((\log n)/2)k_1 = O(\log n)$	$\simeq \frac{nk_2}{2(n-1+\delta)} = O(\log n)$
All-to-all (personalized)	$2(\log n)k_1k_3 + lnk_2$	$2k_3 + 2(n-1+\delta)l$	$\simeq (\log n)k_1 = O(\log n)$	$\simeq \frac{nk_2}{2(n-1+\delta)} = O(\log n)$



(a) Grouping nodes in a column into a cluster



(b) Implementing a *+2 east communication* in two IWR steps

Figure 11: Example of mapping a (4×4) torus onto a 4 processor OMP (P = orthogonal processor).

nication by transforming them into appropriate message vectors.

But, the similarity of a torus structure with that of an orthogonally-accessed memory organization leads to a simplified scheme to emulate these inter-cluster communication steps. Fig 11(b) illustrates an example of implementing a *+2 east communication* in two IWR steps. Let each torus node contains a single data identified by the node numbers. In column-oriented clustering, each orthogonal processor contains n data associated with its column. During the first IWR step, the orthogonal processors, in parallel, perform a *column-write* operation followed by a *row-read* operation. During the second step, the processors, in parallel, do buffer manipulation based on the desired *east* or *west* communication, perform a *row-write* operation with the manipulated data, and then perform a *column-read* operation.

This mapping example provides $k, 1 \leq \mid k \mid \leq n-1$, *east* or *west* communication steps to be implemented with two IWR steps. This leads to a contraction factor of n. The on-the-fly data buffer manipulation logic, as mentioned in Section 3.3, plays an important role in providing this capability. During the *column-write* operation in the first IWR step, the orthogonal processors can *write* manipulated data to the memory modules. This allows up to k column and k row operations, $k, 1 \leq \mid k \mid \leq n-1$, to be combined in two IWR steps. This gives rise to the capability of communication vectorization to map variations of mesh/torus architectures such as mesh with broadcast, mesh with multiple broadcast [12], and generalized mesh [3]. The meshes with broadcast capability can be directly mapped by the OMP due to its flexibility of *broadcast* in the data buffer manipulation logic.

We consider two torus mappings for $m = n$ and $m = n^2$ for analyzing the reduction in communication cost. Saad and Schultz have provided near-optimal algorithms for different communication patterns on a torus. We compare our memory-based emulated communication cost with their results. We use a comparison scheme similar to that developed for hypercube mapping. We denote the fixed and pipelined cost parameters in mesh as t_m and τ_m respectively. Similar to the hypercube mapping analysis, we normalize the parameters as follows:

$$t_o = k_3\tau_o, \quad \tau_m = k_2\tau_o, \quad t_m = k_1 t_o = k_1 k_3 \tau_o \tag{3}$$

Theorem 3 *The mapping of an $(n \times n)$ torus onto an n-processor OMP results in $O(n)$ reduction in communication cost with respect to each of the communication patterns listed in Table 4.*

Table 5: Cost Comparison and the associated Speedup between Torus Communication and Memory-based Communication Vectorization on the OMP (m= number of nodes on target torus architecture, n = number of processors on the host OMP architecture, l = uniform message length on target architecture, k_1, k_2, k_3 = normalizing constants between the fixed and pipelined cost between two schemes, τ = pipelined memory access cost in the OMP).

Communication patterns	Torus communication cost [21] in τ units	Vectorized communication cost in τ units	Communication cost reduction (short message)	Communication cost reduction (long message)
$m = n^2$				
One-to-one	$(n+6)k_1k_3 + \frac{1}{2}k_2$	$k_3 + (1+\delta)l$	$\simeq (n+6)k_1 = O(n)$	$\simeq \frac{k_2}{2(1+\delta)} = O(1)$
One-to-all (broadcast)	$nk_1k_3 + lk_2$	$k_3 + (n-1+\delta)l$	$\simeq nk_1 = O(n)$	$\simeq \frac{k_2}{n-1+\delta} = O(\frac{1}{n})$
All-to-all (broadcast)	$2nk_1k_3 + ln^2k_2$	$k_3 + (n-1+\delta)l$	$\simeq 2nk_1 = O(n)$	$\simeq \frac{n^2k_2}{n-1+\delta} = O(n)$
One-to-all (personalized)	$2nk_1k_3 + ln^2k_2$	$k_3 + (n-1+\delta)nl$	$\simeq 2nk_1 = O(n)$	$\simeq \frac{nk_2}{n-1+\delta} = O(1)$
All-to-all (personalized)	$2nk_1k_3 + 2ln^4k_2$	$k_3 + (n-1+\delta)nl$	$\simeq 2nk_1 = O(n)$	$\simeq \frac{2n^3k_2}{n-1+\delta} = O(n^2)$
$m = n$				
One-to-one	$(\sqrt{n}+6)k_1k_3 + \frac{1}{2}k_2$	$k_3 + (1+\delta)l$	$\simeq (\sqrt{n}+6)k_1 = O(\sqrt{n})$	$\simeq \frac{k_2}{2(1+\delta)} = O(1)$
One-to-all (broadcast)	$\sqrt{n}k_1k_3 + lk_2$	$k_3 + (n-1+\delta)l$	$\simeq \sqrt{n}k_1 = O(\sqrt{n})$	$\simeq \frac{k_2}{n-1+\delta} = O(\frac{1}{n})$
All-to-all (broadcast)	$2\sqrt{n}k_1k_3 + lnk_2$	$k_3 + (n-1+\delta)l$	$\simeq 2\sqrt{n}k_1 = O(\sqrt{n})$	$\simeq \frac{nk_2}{n-1+\delta} = O(1)$
One-to-all (personalized)	$2\sqrt{n}k_1k_3 + lnk_2$	$k_3 + (n-1+\delta)l$	$\simeq 2\sqrt{n}k_1 = O(\sqrt{n})$	$\simeq \frac{nk_2}{n-1+\delta} = O(1)$
All-to-all (personalized)	$2\sqrt{n}k_1k_3 + 2ln^2k_2$	$k_3 + (n-1+\delta)l$	$\simeq 2\sqrt{n}k_1 = O(\sqrt{n})$	$\simeq \frac{2n^2k_2}{n-1+\delta} = O(n)$

Proof: Table 4 analyzes the cost comparison between torus communication and the memory-based emulated communication on the OMP. The cost shown in the table are normalized to τ_o units. Similar to the arguments provided in Theorem 2, the fixed (or variable) costs determine the cost reduction factor for short (or long) message communication. For short messages, the comparison demonstrates $O(n)$ reduction in communication cost. ■

6. Conclusions

The major contribution of this paper lies in demonstrating and developing a systematic mapping method to convert message-passing operations in a multicomputer program into vectorized memory access steps in a shared-memory multiprocessor. The mapping methodology enables efficient and economic program conversion from multicomputers to multiprocessors. In section 5, we have reported the mapping of hypercube and torus programs onto the orthogonal multiprocessor. Continued efforts are in progress to map multicomputer programs onto the bus- and crossbar-connected multiprocessors. New results on these mappings will be included in a revised version of this paper. Comparison of using the three multiprocessor configurations for running the same set of multicomputer programs will be provided then. At this point of time, what we conject is that orthogonal multiprocessor will outperform other multiprocessors, because of using 2-dimensional interleaved memory, instead of 1-dimensional interleaving.

The development of scalable shared-memory multiprocessors with distributed physical memory is approaching to resemble message passing architectures in several ways. Similarly, memory-based interconnection schemes demonstrate potential for being used in a multicomputer environment, because it eliminates the software overhead associated with message-passing communication. Our results in this paper emphasizes on bridging the gap between shared-memory and message-passing models of parallel computing. This opens up new research directions for architecture-independent and communication-independent program development. The proposed mapping method greatly enhances the software portability from multicomputers to multiprocessors.

References

[1] W. C. Athas and C. L. Seitz. Multicomputer: Message-passing Concurrent Computers. *IEEE Computer*, pages 9–25, August 1988.

[2] A. Baratz and K. McAuliffe. A Perspective on Shared-memory and Message-memory Architectures. In J. L. C. Sanz, editor, *Opportunities and Constraints of Parallel Processing*, pages 9–10. Springer-Verlag, 1989.

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

[4] S. Chittor and R. Enbody. Performance Evaluation of Mesh-Connected Wormhole-Routed Networks for Interprocessor Communication in Multicomputers. In *Proceedings of the Supercomputing '90, New York*, pages 647–656, Nov 1990.

[5] K. Gharachorloo. Towards More Flexible Architectures. In J. L. C. Sanz, editor, *Opportunities and Constraints of Parallel Processing*, pages 49–53. Springer-Verlag, 1989.

[6] C. T. Ho and S. L. Johnsson. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transaction on Computers*, 38(9):1249–1268, Sept 1989.

[7] K. Hwang. Exploiting Parallelism in Multiprocessors and Multicomputers. In K. Hwang and D. Degroot, editors, *Parallel Processing for Supercomputers and Artificial Intelligence*, pages 31–67. McGraw Hill, N. Y., 1989.

[8] K. Hwang, M. Dubois, D. K. Panda, S. Rao, S. Shang, A. Uresin, W. Mao, H. Nair, M. Lytwyn, F. Hsieh, J. Liu, S. Mehrotra, and C. M. Cheng. OMP: A RISC-based Multiprocessor using Orthogonal-Access Memories and Multiple Spanning Buses. In *Proc. of ACM International Conference on Supercomputing, Amsterdam, The Netherlands.*, pages 7–22, June 11-15 1990.

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

[10] A. H. Karp. Programming for Parallelism. *IEEE Computer*, pages 43–57, May 1987.

[11] Manoj Kumar. Supporting Broadcast Connections in Benes Networks. Technical Report RC 14063, IBM Research, May 1988.

[12] V. K. Prasanna Kumar and C. S. Raghavendra. Array Processor with Multiple Broadcasting. *Journal of Parallel and Distributed Computing*, 4:173–190, 1987.

[13] Y. Lan, A. H. Esfahanian, and L. M. Ni. Multicast in Hypercube Multiprocessor. *Journal of Parallel and Distributed Computing*, 8:30–41, 1990.

[14] S. Lee and K. G. Shin. Interleaved All-to-all Reliable Broadcast On Meshes and Hypercubes. In *Proceedings of the International Conference on Parallel Processing*, pages III: 110–113, Aug 1990.

[15] X. Lin and L. M. Ni. Multicast Communication in Multicomputer Networks. In *Proc. International Conference on Parallel Processing*, pages III:114–118, 1990.

[16] B. Lint and T. Agerwala. Communication Issues in the Design and Analysis of Parallel Algorithms. *IEEE Transactions on Software Engineering*, SE-7(2):174–188, Mar 1981.

[17] C. Maples. A High-Performance, Memory-Based Interconnection System for Multicomputer Environments. In *Proceedings of the Supercomputing '90, New York*, pages 295–304, Nov 1990.

[18] M. T. O'Keefe and H. G. Dietz. Hardware Barrier Synchronization: Static Barrier MIMD (SBM). In *Proceedings of the International Conference on Parallel Processing*, pages I: 35–42, Aug 1990.

[19] D. K. Panda and K. Hwang. Reconfigurable Vector Register Windows for Fast Matrix Manipulation on the Orthogonal Multiprocessor. In *Proceedings of the International Conference on Application Specific Array Processors, Princeton, New Jersey,*, pages 202–213, Sep 5-7, 1990.

[20] D. K. Panda and K. Hwang. Fast Data Manipulation in Multiprocessors Using Parallel Pipelined Memories. *Journal of Parallel and Distributed Computing, Special Issue on Shared-Memory Systems*, June 1991, in press.

[21] Y. Saad and M. H. Schultz. Data Communication in Parallel Architectures. *Parallel Computing*, 11:131–150, 1989.

[22] J. X. Zhou. A Parallel Computer Model Supporting Procedure-Based Communication. In *Proceedings of the Supercomputing '90, New York*, pages 286–294, Nov 1990.