DISTRIBUTED MEMORY PARALLEL SYSTEMS

A typical parallel computer system consists of a set of computing nodes. Such a computer allows a programmer to divide the computational steps of an application over the set of computing nodes. This division of the computational steps allows the programmer to run the overall application with reduced execution time on a parallel system compared to the time taken on a uniprocessor system. Such division of the computational steps and the associated data of an application to multiple computing nodes is known as *parallel programming*. Two types of parallel programming models are quite common: *distributed memory* and *shared memory*. Parallel computer systems supporting the distributed memory programming model are known as *distributed memory parallel systems*. Similarly, parallel computers supporting shared memory programming model are known as *shared memory parallel systems*.

The two kinds of programming models differ in the way the memory of the computing nodes are made visible to the computing nodes/programmer. Figure 1 shows the distinction. Consider a parallel system consisting of four computing nodes (processors P_0, P_1, P_2, and P_3) and each node having 128 Mbytes of memory (M_0, M_1, M_2, and M_3), as indicated in Fig. 1(a). All together the parallel system has $128 \times 4 = 512$ Mbytes of memory. In a distributed-memory parallel system, each computing node can access its own memory only; that

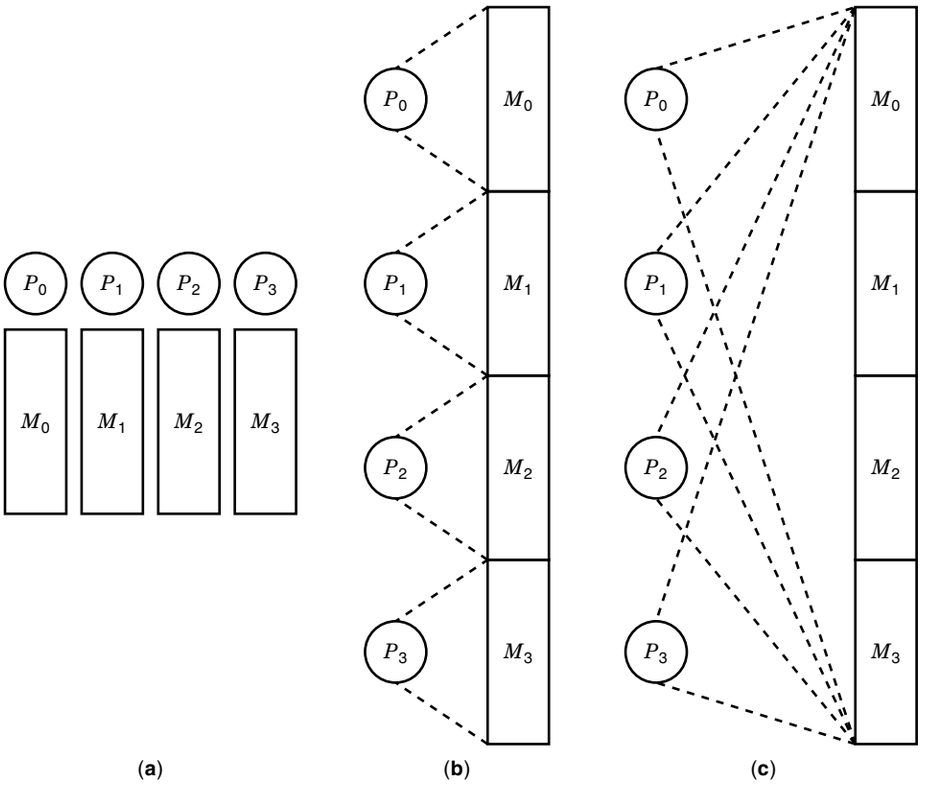

Figure 1. (a) Example of a parallel computer with four computing nodes and associated memory. (b) Distributed memory programming model. (c) Shared memory programming model.

is, P_0 can access only M_0, P_1 can access only M_1, and so on. Such an organization is indicated by Fig. 1(b). However, in a shared memory parallel computer, as shown in Fig. 1(c), each computing node can access the entire 512 Mbytes of memory.

The above two types of parallel systems provide different trade-offs for a programmer when writing a parallel program. It is much easier to write a shared memory parallel program because data can be placed anywhere in the shared memory. However, building such a parallel system delivering very good performance is quite difficult and expensive. Thus, many current generation parallel systems support distributed memory programming model.

In this article, we focus on such distributed memory parallel systems and discuss the associated architectural, communication, programming, and performance issues. We start with the basic concept of message-passing and introduce various communication primitives. Next, we introduce existing message-passing standards and libraries. Basic architectural issues related to the overall system organization are discussed. Issues related to obtaining good performance on such systems are discussed next. Example system architectures corresponding to Intel Paragon, Cray T3E, and IBM SP2 are discussed. Finally, we present future trends of distributed memory parallel systems.

BASIC CONCEPT OF MESSAGE PASSING

The computing nodes of a distributed memory parallel system cannot access each other's memory. However, in order to execute a parallel application, these computing nodes have to exchange information (data and control) among them. Such exchange of information is known as *interprocessor communication* and is achieved by exchanging *messages* between the computing nodes. Each message passing step involves a pairwise operation: a *send* operation from the *sender* computing node and a *recv* operation from the *receiver* computing node.

Figure 2 shows an example of a message passing program involving three computing nodes (P_0, P_1, and P_2). On each computing node, computational steps are interleaved with message passing steps. Processor P_0, after its initial computation phase, sends a message to processor P_1. This message passing step is initiated by a *send* communication primitive. This primitive indicates that a message containing data *datax*

P_0	P_1	P_2
computation	computation	computation
⋮	⋮	⋮
$send(P_1, \text{datax})$	⋮	$send(P_0, \text{dataz})$
⋮	$recv(P_0)$	⋮
⋮	⋮	⋮
computation	computation	computation
⋮	⋮	⋮
$recv(P_2)$	$send(P_2, \text{datay})$	⋮
⋮	⋮	$recv(P_1)$
computation	computation	computation

Figure 2. Example of a message passing program containing three pairs of send–recv message passing steps.

(from the memory of P_0) needs to be sent to processor P_1. Similarly, processor P_1 initiates a *recv* operation to receive the message sent by processor P_0. This example program involves three send-recv pairs of message passing: (P_0, P_1), (P_1, P_2), and (P_2, P_0).

COMMUNICATION PRIMITIVES

Different distributed parallel systems support different kinds of communication primitives to provide flexibility of message passing for the application developers. Broadly, the primitives are divided into two classes: *point-to-point* and *collective.* The first category involves message passing between one sender and one receiver. The second category involves communication between more than two processors. Examples include *broadcast, multicast, barrier synchronization,* and so on (1,2). In *broadcast* operation, one processor sends data to all other processors in the system. Similarly, *multicast* operation involves sending data from one processor to a subset of the other processors. *Barrier synchronization* across a set of processors involves making sure that all processors arrive at a given point in their respective program execution before proceeding further.

Both point-to-point and collective communication primitives are built on top of *send* and *recv* primitives. Different variations of *send* and *recv* primitives also exist (3,4). Some example variations are (1) *synchronous* versus *asynchronous* and (2) *blocking* versus *nonblocking.* A *synchronous send* operation indicates that the processor will not come out of this message passing step unless the message gets delivered to the receiver and an acknowledgment gets returned to the sender. An *asynchronous send* operation does not ensure the acknowledgment step. The completion of such send operation indicates that the message has been sent out from the sender; however, it is not clear whether the receiver has received the message or not. A *blocking recv* operation indicates that the receiver processor must get blocked (not able to proceed) until the message arrives at the receiver. Alternatively, a *nonblocking recv* operation indicates that the receiver processor can return back to the computation if the message does not arrive.

These variations provide different message passing semantics to an application programmer. Depending on the computation–communication characteristics of an application (or for a given part of the application), the programmer can choose to use appropriate communication primitives in order to provide good overlap between computation and communication steps. Such overlap allows the program to run with less time and deliver better parallel speedup.

In addition to the above variations, the communication primitives also differ in the way parameters are passed to these primitives. For example, for a *send* operation, the source location of the data to be sent (from local memory) and its length need to be specified. Similarly, for a *recv* operation, the location where the received data needs to be written at the receiver memory needs to be specified. For a *nonblocking recv* operation, the status of whether it becomes successful or not also needs to be returned to the receiver processor. Such parameter passing together with the above variations provide a wide range of choices of communication primitives for a given distributed memory parallel computer. Readers are requested to refer to Refs. 1, 4, 5, and 6 for details of such variations.

MESSAGE PASSING STANDARDS AND LIBRARIES

Over the years, as different kinds of distributed memory parallel systems got developed, the designers of each system kept on providing a wide range of communication primitives for their own system. This trend allowed a parallel programmer to write the best parallel program for an application on a given system. However, such a practice was found to be very restrictive because the parallel program written for a given system could not be easily ported to another system. The portability was limited because the communication primitives of these two systems were not identical.

For some years, such lack of portability provided a big limitation to the development of parallel programs. This has led to the development of message passing standards. A message passing standard defines a set of communication primitives and its variations to write message passing programs. A parallel system supporting the standard ensures that all communication primitives and their variations are implemented on the system with good performance capability. If a parallel programmer writes a distributed memory parallel program using the communication primitives of the standard, then the program is completely portable across parallel systems supporting the standard. Such standardization allows a great deal of flexibility for application developers to design, develop, and evaluate parallel programs on different parallel systems. It also provides a safeguard for an application developer to be able to run a parallel code (written using the standard) on a newer parallel system when the old one becomes obsolete.

During the last few years, the *message passing interface* (MPI) (4) standard has evolved as the de facto standard for writing message passing programs on distributed memory parallel systems. This standard was developed by a consortium of scientists, engineers, and researchers from parallel computer industry, universities, and research laboratories. The MPI-1 standard was finalized during 1995, and most of the current generation distributed memory parallel systems support this standard. Currently, effort is underway to define the MPI-2 standard with a set of richer communication primitives.

As mentioned earlier, communication primitives belong to two major classes: point-to-point and collective. For a parallel system with a given number of processors, obtaining the best point-to-point communication performance between two nodes for a given data size may require that we use one of the several underlying available implementations. Similarly, implementing any one of the collective communication operations (the MPI standard indicates 14 different collective communication operations) with the best possible performance involves using the best point-to-point communication primitive, using dedicated hardware support for the operation (if available), and using the best communication algorithm (7). The performance of a given collective communication operation also depends on the number of processors and data size used in the operation.

Thus, in order to provide the best choice for an application-level communication (point-to-point or collective), many different implementations need to be available on the system.

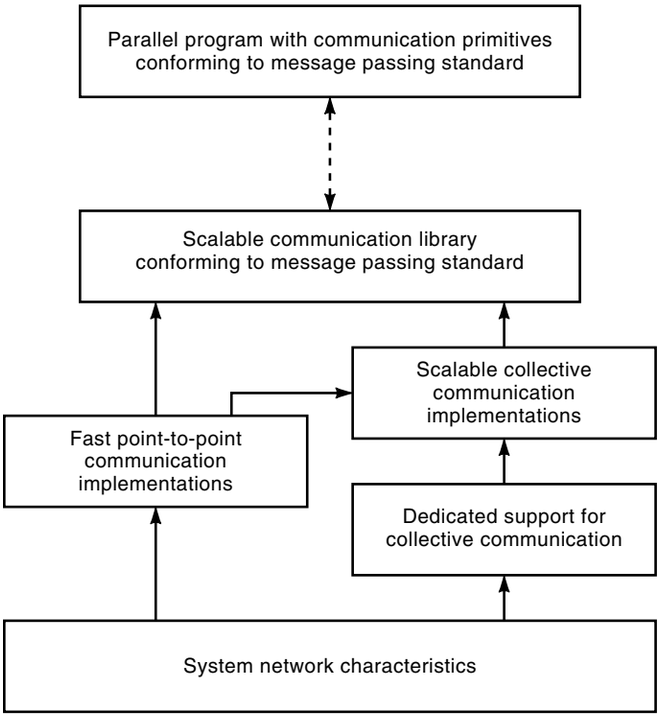

Figure 3. Hierarchies involved in developing a scalable communication library for a distributed memory parallel computer.

Such implementations are typically available as a message passing library form, and the application is linked to the library at compile/run time to select the best possible implementation. Figure 3 shows the typical hierarchical approach used to develop such scalable and high-performance message passing libraries. The point-to-point implementations are developed based on the system/network characteristics. The collective communication implementations are developed based on the point-to-point communication operations and dedicated hardware support for a given collective operation. The scalable communication library is designed by integrating the implementations for point-to-point and collective operations.

Most of the current generation distributed memory parallel systems support scalable communication library conforming to the MPI standard. In addition to this library, some systems also support communication libraries conforming to their own proprietary standards. Examples include the MPL library (8) by IBM SP2, the NX library by Intel Paragon (5), and so on. Such alternative libraries allow flexibility for a parallel programmer to develop an application in a customized manner for a given computer with the best possible performance. These proprietary libraries are also targeted toward providing vertical compatibility with their earlier generation of parallel systems, thus allowing applications written for their earlier generation systems to run with minimum modifications while delivering reasonable performance.

ARCHITECTURAL ISSUES

Major components of designing a distributed memory parallel system are: (a) computing nodes, (b) interconnection network, (c) node-network interfaces, and (d) input/output (I/O) subsystem. The I/O subsystem may be comprised of I/O devices being connected to only a few computing nodes or to all of the computing nodes.

Architectural issues in designing computing nodes for distributed memory parallel systems are the same as those of designing computing nodes for uniprocessor computer systems. The major components of a computing node are high speed microprocessor(s) together with the associated peripheral logic blocks (such as interrupt, direct-memory access [DMA], and timer), main memory blocks, caches, and system bus. Computing nodes for earlier generation distributed memory parallel systems used to be designed in a customized manner. Recently, the trend has changed and the nodes are designed using off-the-shelf microprocessors. Detailed design issues to build high-performance computing nodes for uniprocessor systems are indicated in Ref. 9.

The *interconnection network* and the *node-network interfaces* are significant components of a distributed memory parallel system. The interconnection network typically is used to support interprocessor communication between the computing nodes. If the I/O subsystem is connected to only a few computing nodes then the interconnection network is also used to support I/O traffic.

As the speed of microprocessors continues to increase, the computing nodes are becoming faster. This trend is demanding faster interconnection network (supporting low latency and high bandwidth) so that a given parallel program can be executed faster.

There are several challenges in designing an interconnection network supporting low latency and high bandwidth data transfer. The major components in designing an interconnection network are topology, switching technique, routing, flow control, and switch architecture. There are several choices for each of these components. Some of the choices are dependent on the networking technology. Thus, it is a challenging task for an architect to make appropriate selections to design a high-performance interconnection network. For more information on these design challenges, readers are requested to refer to an accompanying article in this encyclopedia, INTERCONNECTION NETWORKS FOR PARALLEL COMPUTERS.

In addition to designing faster computing nodes and interconnection networks, it is significant to have faster node-network interfaces in a distributed memory parallel system. Otherwise, the communication performance of the system is severely limited. Major components of a node-network interface are a set of *injection channels* to inject messages from the computing node to the interconnection network and a set of *consumption channels* to consume messages from the interconnection network to the computing node. As with designing the interconnection network, there are several challenges in designing a *balanced* node-network interface so that there is no bottleneck in interprocessor communication. Typically, such design involves establishing a close coupling between the network link speed, speed of the computing node, and speed of the I/O bus to which the node-network interface is attached (10). Such a close coupling provides low latency and high bandwidth communication for the system.

The I/O subsystem includes the I/O bus and the I/O devices. As indicated earlier, the I/O subsystem may be attached to only a few nodes or to all nodes of the system. If it is attached to only a few nodes, the system operates in an

asymmetric manner with respect to I/O operation. In such systems, the nodes connected to I/O devices are typically known as *I/O nodes*. The computing nodes take the help of I/O nodes to perform I/O operations. In a symmetric design, all computing nodes are attached with I/O devices. In these systems, I/O operations can be done in parallel by all computing nodes leading to high-performance I/O.

PERFORMANCE ISSUES

The performance of a parallel program on a distributed memory parallel system depends on several factors. Assume that the parallel program is written in an efficient manner, which minimizes communication and I/O steps. The overall execution time of this program will depend on how much overlap can be achieved between the amount of time spent on computation, communication, and I/O. This overlap indirectly depends on several architectural factors previously mentioned. Some of these factors include the speed of computation (processor speed and speed of the memory hierarchy), overhead for communication (operating systems and communication software overheads to inject/consume a message), communication delay in the interconnection network (switch delay, transmission delay, amount of contention in the network), overhead to initiate an I/O operation, and time spent on I/O operation.

Over the years, as the technologies for processor, memory, interconnect, and disk have continued to advance, the parallel computer architects have been involved in designing better and better computing nodes, interconnection networks, node-network interfaces, and I/O subsystems for high-performance distributed memory systems.

While designing these subsystems, architects typically use microbenchmarks to evaluate the performance of the respective subsystems (9,11). Such benchmarking allows them to design the best subsystem under each category. When putting different subsystems together, applications-driven benchmarks (like NAS [12], SPLASH [13]) are typically used to evaluate the performance of a complete system.

ARCHITECTURE OF EXAMPLE SYSTEMS

In this section, we present an architectural overview of some of the current generation distributed memory parallel systems.

Intel Paragon

The Intel Paragon was introduced to the market in 1992 by Intel Corporation as its third-generation distributed memory parallel system (14). Earlier, Intel had introduced the iPSC-1 and iPSC-2 series of distributed memory parallel systems supporting store-and-forward and circuit-switched interconnections, respectively. The Paragon system supported the third-generation wormhole-switched interconnection network (15).

Currently, the largest Paragon system is installed as the Option Red machine at Sandia National Laboratories (16). This system has 4,608 nodes (each consisting of two Pentium Pro processors) with 297 Gbytes of memory. The computing nodes are connected with a $38 \times 32 \times 2$ mesh interconnection network. The system has a peak 1.8 TFLOP computation rate and a peak cross-section bandwidth of over 51 Gbytes/s. The nodes are distributed as follows: 4,536 computing nodes, 32 service nodes, 24 I/O nodes, 2 system nodes, and the remaining hot-spare nodes. In 1997, this system with 9,216 processors was the largest distributed memory parallel system ever built.

Each node in the Paragon system is a shared-memory multiprocessor consisting of two or more processors. One processor works like a dedicated *message* processor. The other processor(s) perform *computation*. Each node is connected to the interconnection network through a *network interface chip* (NIC). Two DMA engines (one for sending messages and the other for receiving messages) are used to support burst data transfer between the memory and network.

The interconnection network for Paragon is a two-dimensional mesh network. A two-plane, two-dimensional network is used in the Option Red system for improved system maintainability. The network supports wormhole-switching and is capable of transmitting data at a peak unidirectional speed of 400 Mbytes/s and 800 Mbytes/s at full duplex.

Cray T3E

The Cray T3E (17,18) system was introduced to the market in 1995. It is a successor to the Cray T3D system and contains several architectural enhancements over the Cray T3D system.

The Cray T3E system is comprised of a number of processing elements (PEs) interconnected by a 3D, bidirectional torus network. This network is primarily used for fast communication. The PEs are also connected by a number of GigaRing channels, which provide connectivity to networks and I/O devices.

Each PE consists of a DEC Alpha 21164 microprocessor, a local memory, a control chip, and a router chip. The control chip provides flexibility for logically shared memory across all PEs: Each PE can access the memory in any other PE, and every PE can access any I/O device through the GigaRing channels.

The T3E supports low-latency, high-bandwidth communication through a 3D torus network. The network supports minimal adaptive routing (18) to minimize network contention for messages. The network is capable of delivering a 64-bit word every system clock (13.3 nsec) in each of all 6 directions. The bisection bandwidth for a 512-PE system exceeds 122 Gbytes/s.

The I/O subsystem of Cray T3E is built with a set of GigaRing channels (19). Each channel is connected to a set of PEs (up to 16 PEs) and to the torus network. Each channel can deliver a peak bandwidth of 1 Gbytes/s. Besides the PEs, other types of I/O nodes and controllers (such as SCSI, FDDI, Ethernet, ATM, and HIPPI) can also be connected to a GigaRing channel. This provides flexibility to provide different kinds of I/O and network connectivity for the system.

IBM Scalable Parallel (SP) System

IBM entered the MPP market with the introduction of the IBM SP1 system in 1993. The SP2 system was introduced in 1994. The SP systems focus on *cluster architecture*. Each node is actually an RS/6000 workstation with its own local disk. A complete AIX (IBM's Unix) resides on each node. A high-

speed interconnection network connects the nodes. The SP systems are designed using custom components as much as possible. This trend brings better systems to the market within short time intervals.

Each node of the IBM SP system consists of a POWER2 processor, local memory, disk, and network connections for the Ethernet and high-performance switch (20). The network connections are through the Micro Channel I/O bus. Depending on the capacity of memory hierarchy, the data path width, and the number of I/O bus slots, IBM SP systems are available with three different node types: *wide node, thin node,* and *thin node 2.*

The nodes are interconnected by two networks: a conventional Ethernet and a high-performance switch. The Ethernet connection is used as a backup interconnection and becomes extremely useful for program development, testing, and debugging. The high-performance switch and its network links are used to support low latency interprocessor communication.

The high-performance switch is a packet-switched, multistage Omega network with buffered wormhole routing (21,22). It is based on the IBM Vulcan crossbar switch chip design. Each chip has eight input and eight output ports. An 8×8 crossbar allows 8 packet cells (called flits) to pass through the switch in every 40 MHz-cycle if there is no conflict. If an output port is already busy then the messages destined for it from input ports get stored in a *central queue* buffer. Thus, this buffering frees up the input ports to receive subsequent flits from the previous switch stage. A set of these switches are interconnected by an additional stage of the network to support up to 128 nodes.

The I/O subsystem is built around the high-performance switch architecture. It has a local area network (LAN) gateway to other machines outside of the SP system. Some of the nodes in a system can work as I/O nodes to perform I/O functions, such as global file servers. Some nodes can also function as *gateway nodes* to serve as networking functions.

CURRENT AND FUTURE TRENDS

The IBM SP architecture has established a new trend for designing cost-effective distributed memory systems: *workstation clusters* or *networks of workstations* (23,24,25). This trend emphasizes using commodity PCs/workstations and commodity interconnects. Currently, several commodity interconnects like Fast Ethernet (26), Gigabit Ethernet (27), ATM (28), and Myrinet (29) are available in the market with varying prices and performance. Thus, depending on the target performance, one or more of these interconnects can be used together with a set of PCs/workstations to design an affordable distributed memory system.

Due to the commodity nature of PCs/workstations and switches, such workstation clusters typically use *irregular* topologies. These topologies are bringing new challenges for developing efficient routing schemes (30), communication mechanisms (31,32), and collective communication algorithms (2). The node-network interfaces on these systems are based on I/O buses (such as PCI for PCs and S-bus for SUN systems). These I/O buses are typically slow and thus limit the overall performance of interprocessor communication. This limitation is leading to development of fast I/O buses and efficient node-network interfaces.

CONCLUSIONS

In this article, we have defined distributed memory parallel systems. Basic concepts of message passing and communication primitives are introduced. Current message passing standards and libraries for distributed memory systems are outlined. Main architectural and performance issues are discussed. Architectures of a set of representative distributed memory systems (such as Intel Paragon, Cray T3E, and IBM SP) are presented. Finally, current and future trends in designing high-performance and cost-effective distributed memory systems are outlined.

This article has attempted to provide an overview of distributed memory systems. Interested readers are requested to refer to additional readings (33,34,35) for more information.

BIBLIOGRAPHY

1. P. K. McKinley and D. F. Robinson, Collective communication in wormhole-routed massively parallel computers, *IEEE Comput.,* pp. 39–50, Dec. 1995.
2. D. K. Panda, Issues in designing efficient and practical algorithms for collective communication in wormhole-routed systems, *ICPP Workshop on Challenges for Parallel Process.,* 1995, pp. 8–15.
3. R. Brightwell and A. Skjellum, MPICH on the T3D: A case study of high-performance message passing, *Proc. 2nd MPI Develop. Conf.,* 1996, pp. 2–9.
4. Message passing interface forum. *MPI: A Message-Passing Interface Standard,* Mar. 1994.
5. P. Pierce and G. Regnier, The Paragon implementation of the NX message passing interfaces, *Proc. Scalable High-Perform. Comput. Conf.,* May 1994, pp. 184–190.
6. V. S. Sunderam, PVM: A framework for parallel and distributed computing, *Concurrency: Practice and Experience,* **2** (4): 315–339, December 1990.
7. D. K. Panda, Fast barrier synchronization in wormhole k-ary n-cube networks with multidestination worms, *Future Generation Comput. Syst.,* **11**: 585–602, Nov. 1995.
8. Z. Xu and K. Hwang, Modeling communication overhead: MPI and MPL performance on the IBM SP2, *IEEE Parallel and Distrib. Technol.,* pp. 9–23, Spring 1996.
9. J. L. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* 2nd ed., San Mateo, CA: Morgan Kaufmann, 1996.
10. D. Basak and D. K. Panda, Alleviating consumption channel bottleneck in wormhole-routed k-ary n-cube systems. *IEEE Trans. Parallel Distrib. Syst.,* **9**, 1998.
11. C. Hristea, D. Lenoski, and J. Keen, Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks, *ACM Supercomput. Conf. (SC97),* October 1997.
12. D. H. Bailey et al., NAS parallel benchmark results. Technical Report 94-006, RNR, 1994.
13. S. C. Woo et al., The SPLASH-2 programs: Characterization and methodological considerations, *Int. Symp. Comput. Archit.,* 1995, pp. 24–36.
14. Intel Corporation. *Paragon XP/S Product Overview,* 1991.
15. L. Ni and P. K. McKinley, A survey of wormhole routing techniques in direct networks, *IEEE Comput.,* pp. 62–76, Feb. 1993.

16. T. G. Mattson, D. Scott, and S. Wheat, A TeraFLOP supercomputer in 1996: The ASCI TFLOP system, *Proc. 10th Int. Parallel Process. Symp.,* 1996, pp. 84–93.
17. S. L. Scott, Synchronization and communication in the T3E multiprocessor, *Proc. 7th Int. Conf. Arch. Support Program. Lang. Operat. Sys.,* 1996, pp. 26–36.
18. S. L. Scott and G. M. Thorson, The Cray T3E network: Adaptive routing in a high performance 3D torus, *Proc. Symp. High Perform. Intercon. (Hot Intercon. 4),* August 1996, pp. 147–156.
19. S. L. Scott, The GigaRing channel. *IEEE Micro.,* February 1996, pp. 27–34.
20. T. Agerwal et al., SP2 system architecture, *IBM Syst. J.,* **34** (2): 263–272, 1995.
21. C. B. Stunkel et al., The SP1 high performance switch. *Scalable High Perf. Comput. Conf.,* 1994, pp. 150–157.
22. Craig B. Stunkel et al., The SP2 high-performance switch. *IBM Syst. J.,* **34** (2): 185–204, 1995.
23. T. Anderson, D. Culler, and Dave Patterson, A case for networks of workstations (NOW). *IEEE Micro.,* pp. 54–64, Feb. 1995.
24. D. K. Panda and C. B. Stunkel (eds.), *Communication and Architectural Support for Network-Based Parallel Computing (CANPC), Lecture Notes in Computer Science,* Volume 1199, New York: Springer-Verlag, 1997.
25. D. K. Panda and C. B. Stunkel (eds.), *Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC), Lecture Notes in Computer Science,* Volume 1362, New York: Springer-Verlag, 1998.
26. L. Melatti, Fast Ethernet: 100 Mbit/s made easy, *Data Commun.* [on the Web], (http://www.data.com/tutorials/100mbits_made_easy.html), Nov. 1994.
27. Craig Partridge, *Gigabit Networking,* Reading, MA: Addison-Wesley, 1994.
28. ATM forum. *ATM User-Network Interface Specification, Version 3.1,* September 1994.
29. N. J. Boden et al., Myrinet: A gigabit-per-second local area network. *IEEE Micro.,* pp. 29–35, Feb. 1995.
30. W. Qiao and L. M. Ni, Adaptive routing in irregular networks using cut-through switches, *Proc. Int. Conf. Parallel Process.,* Chicago, Aug. 1996, pp. 52–60.
31. S. Pakin, M. Lauria, and A. Chien, High performance messaging on workstations: Illinois fast messages (FM), *Proc. Supercomput.,* 1995.
32. T. von Eicken et al., U-Net: A user-level network interface for parallel and distributed computing, *ACM Symp. Oper. Syst. Princ.,* 1995.
33. D. E. Culler and J. P. Singh, *Parallel Computer Architecture: A Hardware-Software Approach,* San Mateo, CA: Morgan Kaufmann, 1998.
34. J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach,* Los Alamitos, CA: IEEE Computer Society Press, 1997.
35. K. Hwang and Z. Xu, *Scalable Parallel Computing,* New York: McGraw Hill, 1998.

DHABALESWAR K. PANDA
The Ohio State University