The Paradigm Compiler for Distributed-Memory Multicomputers

Prithviraj Banerjee
John A. Chandy
Manish Gupta
Eugene W. Hodges IV
John G. Holm
Antonio Lain
Daniel J. Palermo
Shankar Ramaswamy
Ernesto Su

University of Illinois,
Urbana-Champaign

Massively parallel distributed-memory multicomputers can achieve the high performance levels required to solve the Grand Challenge computational science problems (a class of computational applications, identified by the 1992 US Presidential Initiative in High-Performance Computing and Communications, that would require a significant increase in computing power). Multicomputers such as the Intel Paragon, the IBM SP-1/SP-2 (Scalable Power Parallel 1 and 2) and the Thinking Machines CM-5 (Connection Machine 5) offer significant cost and scalability advantages over shared-memory multi-processors. However, to harness these machines' computational power, users must write efficient software. This process is laborious because of the absence of global address space. The programmer must manually distribute computations and data across processors and explicitly manage communication. The Paradigm (Parallelizing Compiler for Distributed-Memory, General-Purpose Multicomputers) project at the University of Illinois addresses this problem by developing automatic methods for efficient parallelization of sequential programs.

THE PARADIGM ADVANTAGE

To demonstrate the complexity of writing programs in a message-passing model, we examine a small sample of compiler-generated parallel code (see Figure 1), which roughly corresponds to what an experienced programmer might write. Figure 1a presents a sample serial program for Jacobi's iterative method of solving systems of linear equations. Figure 1b shows a highly efficient parallel version of this program generated for an Intel Paragon. From this example, it is apparent that if a programmer were required to manually parallelize even a moderately sized application, the effort would be tremendous. Furthermore, coding with explicit communication operations commonly results in errors that are notoriously hard to find. By automating the parallelization process, we can offer high levels of performance to the scientific computing community at large.

Other research efforts in this area include the Fortran D1 and Superb compiler. In a collaborative effort of industry and academia, researchers have also developed High-Performance Fortran (HPF) to standardize parallel programming with data distribution directives. In addition, several commercial HPF compilers are emerging from Applied Parallel Research, Convex, Digital, IBM, the Portland Group, Thinking Machines, and others.

However, Paradigm addresses a broad range of research topics in the scope of a single framework, setting it apart from other compiler efforts.

for distributed-memory multicomputers. Research in the Paradigm project aims to automate the selection of data distributions, optimize communication and distribute regular computations, support irregular computations using a combination of compile-time analysis and runtime support, exploit functional and data parallelism simultaneously, and generate multithreaded message-driven code to hide communication latencies. Current efforts aim at integrating all of these capabilities into the Paradigm framework.

**COMPILER FRAMEWORK**

Figure 2 shows how we envision the complete Paradigm compilation system. The compiler currently accepts either a sequential Fortran 77 or HPF program and produces an explicit message-passing version—a Fortran 77 program with calls to the selected message-passing library and our runtime system. The phases in the parallelization process are also shown along with their interactions.

Program analysis

Parafnse-2 acts as a preprocessing platform that parses the sequential program into an intermediate representation and analyzes the code to generate flow, dependence, and call graphs. This stage also includes various code transformations, such as constant propagation and induction variable substitution.

Automatic data partitioning

For regular computations, the compiler can automatically determine the best static distribution of program data. The compiler configures the machine as an abstract multidimensional mesh of processors and determines how program data is distributed on this mesh. Estimates of computation and communication time, reflecting high-level communication operations and other communication optimizations performed in the compiler, are used to correctly determine the best distribution.

Program:

```
program jacobi
character m$buf(1000)
integer m$sp(2), m$BA1, m$BA2, m$B1, m$B2
integer m$griddim, m$num(2), m$to(-1:1, -1:1), m$inc
real A, B(0:251,0:251)
m$griddim = 2
m$num(1) = 2
m$num(2) = 2
m$gridinit(m$numdim, m$num, m$to(-1:1, -1:1), m$inc)
real A(250, 250), B(0:251, 0:251)
A(i,j) = (B(i-1, j) + B(i+1, j) + B(i, j-1) + B(i, j+1)) / 4
end if
end do
end do
end do
end do
end
```

Figure 1. Sample program, Jacobi’s iterative method: (a) serial version; (b) optimized parallel version.
Regular computations
Using the owner-computes rule, the compiler statically partitions regular computations across processors according to the selected data distribution and generates interprocessor communication for required nonlocal data. To avoid the overhead of computing ownership at runtime, static analysis is used to partition loops at compile time. In addition, several optimizations are employed to reduce the overhead of communication.6

Irregular computations
In many important applications, compile-time analysis is insufficient when communication patterns are data dependent and known only at runtime. In a subset of these applications, the communication pattern repeats in several steps. Paradigm approaches these problems through a combination of flexible, irregular runtime support and compile-time analysis. Novel features in our approach are the exploitation of spatial locality and the overlapping of computation and communication.7

Functional parallelism
For applications with insufficient data parallelism and some functional parallelism, our research8 has shown the benefits of simultaneously exploiting data and functional parallelism. Such applications can be viewed as a graph composed of data-parallel tasks with precedence relationships describing the functional parallelism that exists among those tasks. Using this task graph, Paradigm exploits functional parallelism by determining the number of processors to allocate for each data-parallel task and scheduling the tasks to minimize overall execution time. The same techniques used for regular and irregular data-parallel compilation are also used to generate code for each task.

Multithreading
At this point, the compiler has generated a message-passing program that sends messages asynchronously and blocks when waiting for messages, but in some cases, this can be inefficient. One solution is to run multiple threads on each processor to overlap computation and communication.9 Multithreading lets one thread use the unused cycles that would otherwise be wasted waiting for messages. Compiler transformations convert message-passing code into a message-driven model to simplify the multithreading runtime system. Multithreading is most beneficial for programs with a high percentage of idle cycles where the overhead of switching between threads can be hidden.

Generic library interface
Support for specific communication libraries is provided

Distributed-memory compiler terminology
Data parallelism—Parallelism that exists via similar operations performed simultaneously across different elements of a data set: SPMD (single program, multiple data).
Functional parallelism—Parallelism that exists via potentially different operations performed on different data sets simultaneously: MPMD (multiple program, multiple data).
Regular computations—Computations that typically use dense (regular) matrix structures (regular accesses can usually be characterized using compile-time analysis).
Irregular computations—Computations that typically use sparse (irregular) matrix structures (irregular accesses are usually input data dependent requiring runtime analysis).
Data partitioning—The physical distribution of data across the processors of a parallel machine to efficiently use available memory and improve the locality of reference in parallel programs.

Global index/address—Index used to access an element of an array dimension when the entire dimension is physically allocated on the processor (equivalent to the index used in a serial program).
Local index/address—Index pair (processor, index) used to access an element of an array dimension when the dimension is partitioned across multiple processors (the local index can also refer to only the index portion of the pair).
Owner-computes rule—States that all computations modifying the value of a data element are to be performed by the processor to which the element is assigned by the data partitioning.
User-level thread—A context of execution under user control that has its own stack and registers.
Multithreading—A set of user-level threads that share the same user data space and cooperatively execute a program.
through a generic library interface. For each supported library, abstract functions are mapped during compile time to corresponding library-specific code generators. Library interfaces have been implemented for the Thinking Machines CMMD (Connection Machine MIMD (multiple instruction, multiple data)), Parasoft Express, MPI (Message-Passing Interface), Intel NX (Node Executive), PVM (Parallel Virtual Machine), and PICL (Portable Instrumented Communication Library). Execution tracing, as well as support for multiple platforms, is also provided in Express, MPI, PVM, and PICL. The portability of this library interface lets the compiler easily generate code for a wide variety of machines.

AUTOMATIC DATA PARTITIONING

Determining the best data partitioning for an application is a difficult task requiring careful examination of numerous tradeoffs. Since communication tends to be more expensive relative to local computation, the selected partitioning should maintain high data locality for each processor. Excessive communication can easily offset any gains made via available parallelism in the program. At the same time, the partitioning should evenly distribute the workload among the processors, making full use of the parallelism present in the computation. The programmer might not be—and should not have to be—aware of all interactions between distribution decisions and compiler optimizations. By performing automatic data partitioning, the compiler

• reduces the burden on the programmer,
• improves program portability and machine independence, and
• improves the selection of data distributions.

There is often a tradeoff between minimizing interprocessor communication and exploiting all available parallelism; the communication and computational costs imposed by the underlying architecture must be considered. These costs are generated via architectural parameters for each target machine. Except for architecture-specific costs, the partitioning algorithm is machine independent.

In the compiler, data partitioning decisions are made in distinct phases (see Figure 3). In each phase performed during the partitioning pass, a detector module identifies data distribution preferences, a driver assigns costs to quantify the estimated performance impact of those preferences, and a solver resolves any conflicts.

Array alignment

The alignment pass identifies which array dimensions should be mapped to the same processor mesh dimension. The alignment preferences between two arrays can involve different pairings of dimensions (interdimensional alignment) or can be specified to be an offset (data shift) or stride (data compression) within a given pair of dimensions (intradimensional alignment). Currently, only interdimensional alignment is performed in the partitioning pass.

Block/cyclic distribution

After array alignment, the distribution pass determines whether each array dimension requires a blocked or cyclic distribution (see sidebar “Data distribution”). Array dimensions are first classified by their communication requirements. A nearest-neighbor communication pattern in a mesh dimension requires a blocked distribution. For dimensions that are only partially traversed (less than a certain threshold), a cyclic distribution might be better for load balancing. Using alignment information from the previous phase, the array dimensions that cross-reference one another are also assigned the same kind of partitioning to ensure the intended alignment.

Block size selection

When a cyclic distribution is chosen, communication costs must be examined closely to improve the load balance for partially traversed array dimensions. The compiler can make further adjustments on the block size, generating block/cyclic partitionings. This analysis is sometimes needed when arrays are used to simulate record-like structures (not supported directly in Fortran 77) or when lower dimensional arrays play the role of higher dimensional arrays.

Mesh configuration

Once all the distribution parameters have been determined, the cost estimates are now only functions of the number of processors in each mesh dimension. For each set of aligned-array dimensions, the compiler identifies any parallel operations. If no parallelism exists in a given dimension, that dimension is collapsed onto a single processor. If only one dimension has not been collapsed, all processors are
assigned to that dimension. For multiple dimensions of parallelism, the compiler determines the best arrangement of processors by evaluating the cost expression to estimate execution time for each feasible configuration. At this point, the distribution information is passed on to direct the remainder of the parallelization process in the compiler. The user can also generate an HPF program containing the directives that specify the selected partitioning. Hence, the partitioning pass can be used as an independent tool while remaining integrated with the compilation system, ensuring that the data partitioning pass is always aware of the compiler's optimizations. Paradigm can further improve performance by selecting points in the program at which it would be beneficial to let the data be redistributed. We are currently extending the static partitioner to automatically determine when such dynamic data partitionings are useful.

**REGULAR COMPUTATIONS**

For regular computations where the communication pattern can be determined at compile time, Paradigm uses static analysis to partition computation across processors and to optimize interprocessor communication. For efficient analysis, the compiler needs a mechanism that describes partitioned data and iteration sets. Processor tagged descriptors (PTDs) uniformly represent these sets for every processor. Set operations on PTDs are extremely efficient, simultaneously capturing the effect of partitioned data and iteration sets for all processors in a given dimension.

PTDs, however, are not general enough to handle extremely complicated array distributions, references, or loop bounds occasionally found in real code (see Figure 4). Paradigm represents partitioned data and iteration sets by symbolic linear inequalities and generates loops to scan these regions using a technique known as Fourier-Motzkin projection. To implement Fourier-Motzkin projection, Mathematica (Wolfram Research's powerful off-the-shelf symbolic analysis system) is linked with the compiler to provide a high level of symbolic support and rapid prototyping. Thus, by using the efficient PTD representation for the simplest and most frequent cases and a more general, inequality-based representation for the difficult cases, Paradigm can compile a larger proportion of programs without jeopardizing compilation speed.

The performance of the resulting parallel program also greatly depends on how well its interprocessor communication has been optimized. Since the start-up communication cost (overhead) tends to be several orders of magnitude greater than either the per-element computation cost or the per-byte transmission cost (rate), frequent communication can easily dominate the execution

**Data distribution**

Arrays are physically distributed across processors to efficiently use available memory and improve the locality of reference in parallel programs. In High-Performance Fortran, either the programmer or the compiler must specify the distribution of program data. Figure A shows several examples of data distributions for a two-dimensional array. Each dimension of an array can be given a specific distribution. Blocked and cyclic distributions are actually two extremes of a general distribution commonly referred to as block-cyclic (or cyclic(k), where k is the block size). A block distribution is equivalent to a block-cyclic distribution in which the block size is the size of the original array divided by the number of processors, cyclic(NP). A cyclic distribution is simply a block-cyclic distribution with a block size of one, cyclic(1). Dimensions that are not partitioned across the processors are considered to be “collapsed” in the processor space.
A linear point-to-point transfer cost for a message of m bytes is used as a basis for the communication model:

\[
\text{transfer (m)} = \text{overhead} + \text{rate} \times m
\]

where the values for the overhead and rate are empirically measured for a given machine.

Several optimizations combine messages in different ways to amortize the start-up cost and thereby reduce the total amount of communication overhead in the program. In loops with no cross-iteration dependencies, Paradigm extracts parallelism by independently executing groups of iterations on separate processors. For independent references, the overhead associated with frequent communication is reduced through message coalescing, message vectorization, and message aggregation. For references within loops that contain cross-iteration dependencies, coarse-grain pipelining optimizes communication across loops while balancing the overhead with the available parallelism.

**Message coalescing**

Redundant communication for different references to the same data is unnecessary if the data has not been modified between uses. When statically analyzing individual references, Paradigm detects redundant communication and coalesces it into a single message, letting the data be reused rather than communicated for every reference. For different sections of a given array, Paradigm coalesces individual elements by unifying the different sections, thereby ensuring that overlapping data elements are communicated only once. Since coalescing will either eliminate entire communication operations or reduce the size of messages containing array sections, it is always beneficial.

**Message vectorization**

Nonlocal array elements indexed within a loop nest can be vectorized into a single, larger message instead of communicated individually (see Figure 5, top). Dependence analysis determines the outermost loop for which the communication from a given reference can be vectorized. The element-wise messages are combined, or vectorized, as they are lifted from the enclosing loop nests to the selected vectorization level. Vectorization reduces the number of communication operations (hence, the total overhead) while increasing the message length.

**Message aggregation**

Multiple messages communicated between the same source and destination can also be aggregated into a single larger message. Communication operations are first sorted by their destinations during the analysis. Messages with identical source and destination pairs are then combined into a single communication operation (see Figure 5, bottom). Aggregation can be performed on communication operations of individual data references as well as vectorized communication operations. The gain from aggregation is similar to vectorization in that total overhead is reduced but message length is increased.

Figure 6 illustrates the efficacy of these optimizations, showing the performance of several program fragments executed on a 64-processor CM-5. The automatic data-distribution pass selected linear (1D) partitionings for both alternating-direction implicit (ADI) integration (Livermore kernel 8 with 16,000-element arrays) and explicit hydrodynamics (Livermore kernel 18 with 16,000-element arrays). The distribution selected a 2D partitioning for Jacobi's iterative method (similar to that previously shown in Figure 1 with 1,000 x 1,000 matrices).

For comparison purposes, the reported execution times have been normalized to the serial execution of the corresponding program and are further separated into two quantities:

- **Busy**: Time spent performing the actual computation.
- **Overhead**: Time spent executing code related to computation partitioning and communication.

You can see the relative effectiveness of each optimization by examining the amount of overhead eliminated as the optimizations are incrementally applied.

Moreover, an additional run of a 1D partitioned version of Jacobi has a higher overhead than the compiler-selected 2D version. This shows the effectiveness of the automatic data-partitioning pass, since it could select the best distribution despite minor performance differences. For larger machine sizes and more complex programs, the utility of automatic data distribution is even more apparent, as communication costs increase for inferior data distributions.

**Coarse-grain pipelining**

When there are cross-iteration dependencies due to recurrences, Paradigm cannot immediately execute every iteration in parallel. In many cases, however,
IRREGULAR COMPUTATIONS

Compile-time analysis is insufficient when the required communication patterns are data dependent and, thus, known only at runtime. For example, the computation of airflow and surface stress over an airfoil might use an irregular finite-element grid. To efficiently run such irregular applications on a massively parallel multicomputer, runtime compilation techniques can be used. The program's dependency structure is analyzed in a preprocessing step before the actual computation occurs. If the same computation structure is maintained across several steps, this preprocessing can be reused to amortize cost. In practice, this concept is implemented with two sequences of code: an inspector for preprocessing and an executor for performing the actual computation.

The preprocessing step performed by the inspector can be very complex: The unstructured grid is partitioned, the resulting communication patterns are optimized, and global indices are translated into local indices. During the executor phase, elements are communicated based on this preprocessing analysis. To simplify the implementation of inspectors and executors, irregular runtime support (IRTS) is used to provide primitives for these operations.
There are several ways to improve a state-of-the-art IRTS such as Chaos/Parti (Parallel Automated Runtime Toolkit). The internal representation of communication patterns in such systems is somewhat restricted. These systems represent irregular patterns that are completely enumerated or regular block patterns. Neither system optimizes both regular and irregular accesses, and neither efficiently supports the small regular blocks that arise in irregular applications written to exploit spatial cache locality. Moreover, systems such as Chaos/Parti do not provide nonblocking communication primitives that can further increase performance.

All of these problems are addressed in the Parallel Irregular Library with Application of Regularity (PILAR), Paradigm's IRTS for irregular computations. PILAR is written in C++ to easily support different internal representations of communication patterns. Hence, using a common framework, Paradigm can efficiently handle a wide range of applications, from fully irregular to regular. PILAR uses intervals to describe small regular blocks and enumeration to describe patterns with little or no regularity. The object-oriented nature of the library simplifies both the implementation of new representations and the interactions among objects having different internal representations.

We conducted an experiment to evaluate the effectiveness of PILAR in exploiting spatial regularity in irregular applications. After a partitioner had assigned nodes to processors, we measured the overhead to redistribute the edges of an unstructured grid. We assumed a typical CSR (compressed sparse row) or Harwell-Boeing initial layout (where edges of a given grid node are contiguous in memory). Redistribution occurs in two phases: The inspector phase computes a schedule that captures the redistribution of the edges and sorts the new global indices, and the executor phase redistributes the array with the previously computed schedule using a global data-exchange primitive.

The experiment used a large, unstructured grid from NASA called Rotor. A large ratio (9.40:1) between the maximum and average degree of a node in this grid would make a two-dimensional matrix representation of the edges inefficient. Chaos/Parti was compared against PILAR with both enumeration and intervals during the two phases of the redistribution. Results for a 32-processor IBM SP-1 appear in Figure 8, which clearly shows the benefit of using the more compact interval representation. Further experiments also showed that only three edges per grid node are required to benefit from an interval-based representation in the SP-1.

Even with adequate IRTS, generating efficient inspector or executor code for irregular applications is fairly complex. In Paradigm, compiler analysis for irregular computations is used to detect preprocessor reuse, insert communication primitives, and highlight opportunities for exploiting spatial locality. After performing this analysis, the compiler generates inspector and executor code via embedded calls to PILAR routines.

**FUNCTIONAL AND DATA PARALLELISM**

The efficiency of data-parallel execution tends to decrease as the number of processors increases for a given problem size or as problem size decreases for a given number of processors. By exploiting functional as well as data parallelism, we can sometimes improve a program's overall execution efficiency. A task graph, known as a macro dataflow graph (MDG), represents the functional and data parallelism available in a program. This graph is a weighted directed-acyclic graph (DAG) with nodes representing data-parallel routines in the program and edges representing precedence constraints among those routines. In the MDG, data parallelism is implicit in the nodes' weight functions, while functional parallelism is captured by the precedence constraints among nodes.

Node and edge weights stem from processing and data redistribution costs. The processing cost is the computation and communication time required to execute a data-parallel routine and depends on the number of processors used to execute the routine. Scheduling may make it necessary to redistribute an array between the execution of a pair of routines. The time required for this data redistribution depends on the number of processors and the data distributions used by those routines.

An allocation and scheduling approach on the MDG determines the best execution strategy for a given program. Allocation determines the number of processors to use for each node, while scheduling yields an execution scheme for the allocated nodes on the target multicomputer. Figure 9 (top) shows an MDG with three nodes (N₁, N₂, and N₃) along with their processing costs and efficiencies as a function of the number of processors they use. For this example, we assume no data redistribution costs exist among the three routines. Figure 9 (bottom) shows two execution schemes for a four-processor system. The first scheme exploits pure data parallelism, with all routines using four processors. The second scheme exploits both functional and data parallelism, with routines N₁ and N₂ executing concurrently and using two processors each. As shown, good allocation and scheduling can decrease program execution time.

Paradigm's allocation and scheduling algorithms are based on the mathematical forms of the processing and data-redistribution cost functions; these functions belong to a class known as posynomials. Paradigm uses them to for-
mulate the problem via convex programming for optimal allocation. After allocation, Paradigm uses a list-scheduling policy to schedule the nodes on a given system. The finish time obtained via this scheme is within a factor of the theoretical optimal finish time; in practice, this factor is small. Figure 10 compares the performance of the allocation and scheduling approach to that of the pure data-parallel approach. Speedup values are computed for the Paragon and CM-5 for a pair of applications. Performance using the allocation and scheduling approach is identified as MPMD (multiple program, multiple data), and performance for the pure data-parallel scheme is called SPMD (single program, multiple data). The first application shown is Strassen's matrix multiplication algorithm. The second is a computational fluid-dynamics code using a spectral method. For machines with many processors, the performance of MPMD relative to SPMD execution is improved by a factor of two or three. These results demonstrate the utility of the allocation and scheduling approach.

MULTITHREADING

When the resulting parallel program has a high percentage of idle cycles, multithreading can further improve performance. Running multiple threads on each processor, one of the threads can use the cycles that would otherwise be wasted waiting for messages. To support multithreaded execution, the compiler first generates message-passing code for more virtual processors than physical processors in a given machine. The compiler then maps multiple virtual processors onto physical processors, generating multiple execution threads for each physical processor.

To execute multithreaded code efficiently, Paradigm uses compiler transformations to convert message-passing code into a message-driven model, thereby simplifying the multithreading runtime system (MRTS). The message-driven model uses receive operations to switch between threads and therefore must return control to the MRTS. The transformation required is simple for code without conditionals but becomes more complex when conditionals and loops are included. Although we are only presenting the transformation for converting while loops to message-driven code, similar transformations can be performed on other control structures.

Figure 9. Example of functional parallelism: (top) macro dataflow graphs for program with tasks \( N_1 \), \( N_2 \), and \( N_3 \) and processing costs and efficiencies for those tasks; (bottom) allocation and scheduling for SPMD and MPMD.

Figure 11 shows the transformation of the while loop. The left side shows the control-flow graph of a message-passing program containing a receive in a while loop. The middle and right sides show the transformed code, where main1 is constructed such that code A is executed followed by the while condition check. If the while loop condition is true, code B is executed, the receive is executed, the routine enables main2, and main1 returns to the MRTS to execute other threads. If the while loop condition is false, code D is executed. If main2 is enabled, it will receive its message and execute code C, which is the code after the receive, inside the while loop. At this point, the original code would check for loop completion. Therefore, the transformed code must also perform this check; if loop completion is true, main2 enables another invocation of itself and returns to the MRTS. Otherwise,
PARADIGM IS A FLEXIBLE PARALLELIZING COMPILER for multi-computers. It can automatically distribute program data, perform various communication optimizations for regular computations, and support irregular computations using compilation and runtime techniques. For programs with functional parallelism, the compiler can increase performance through proper resource allocation and scheduling. Paradigm can also use multithreading to further increase the efficiency of codes by overlapping communication and computation. When all of these methods are fully integrated, Paradigm will be able to compile a wide range of applications for distributed-memory multicomputers.

Acknowledgments
This research was supported in part by the Office of Naval Research under Contract N00014-91-J-1096, the National Aeronautics and Space Administration under Contract NASA NAG 1-613, an AT&T graduate fellowship, a Fulbright/MEC (Ministerio de Educacion y Ciencia) fellowship, an IBM graduate fellowship, and an Office of Naval Research (ONR) graduate fellowship. We are also grateful to the National Center for Supercomputing Applications, the San Diego Supercomputing Center, and the Argonne National Laboratory for providing access to their machines.

References
2. B. Chapman, P. Mehrotra, and H. Zima, “Programming in
Prithviraj Banerjee is the director of computational science and engineering and a professor of electrical and computer engineering at the University of Illinois, Urbana-Champaign. His research interests include distributed-memory multicomputer architectures and compilers, and parallel algorithms for VLSI CAD. Banerjee received a BTech degree in electronics and electrical engineering from the Indian Institute of Technology, Kharagpur, India, in 1981 and MS and PhD degrees in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1982 and 1984, respectively.

John A. Chandy is a research assistant in the Center for Reliable and High-Performance Computing at the University of Illinois, Urbana-Champaign. He received an SB degree in electrical engineering from MIT in 1989 and an MS degree in electrical engineering from the University of Illinois in 1993. He is pursuing a PhD degree at the University of Illinois.

Manish Gupta works at the IBM T.J. Watson Research Center, Yorktown Heights, New York. He received a BTech degree in computer science from the Indian Institute of Technology, Delhi, in 1987, an MS degree in computer and information science from Ohio State University, Columbus, in 1988, and a PhD degree in computer science from the University of Illinois, Urbana-Champaign, in 1992.

Eugene W. Hodges IV is a research assistant in the Center for Reliable and High-Performance Computing at the University of Illinois, Urbana-Champaign. He received a BS in computer engineering from North Carolina State University in 1993 and an MS degree in electrical engineering from the University of Illinois in 1995. He is pursuing a PhD degree at the University of Illinois.

John G. Holm is a research assistant in the Center for Reliable and High-Performance Computing at the University of Illinois, Urbana-Champaign. He received a BS degree in electrical engineering from the University of Michigan in 1989 and an MS degree in electrical engineering from the University of Illinois in 1992. He is pursuing a PhD degree at the University of Illinois.

Antonio Lain is a doctoral candidate in the Department of Computer Science at the University of Illinois, Urbana-Champaign. He received a BS degree in ingeniero de telecomunicacion from the Universidad Politecnica de Madrid in 1990 and an MS degree in computer science from the University of Edinburgh, Scotland, in 1991.

Daniel J. Palermo is a research assistant in the Center for Reliable and High-Performance Computing at the University of Illinois, Urbana-Champaign. He received a BS degree in computer and electrical engineering from Purdue University, West Lafayette, in 1990 and an MS degree in computer engineering from the University of Southern California in 1991. He is pursuing a PhD degree at the University of Illinois.

Shankar Ramaswamy is a doctoral candidate in the Department of Electrical Engineering at the University of Illinois, Urbana-Champaign. He received a BS degree in electronics from the University of Delhi, New Delhi, in 1987 and an ME degree in electrical engineering from the Indian Institute of Science, Bangalore, in 1991.

Ernesto Su is a research assistant in the Center for Reliable and High-Performance Computing at the University of Illinois, Urbana-Champaign. He received a BS degree in electrical engineering from Columbia University, New York, in 1980 and an MS degree in electrical engineering from the University of Illinois in 1993. He is pursuing a PhD degree at the University of Illinois.

Readers can contact Banerjee at the University of Illinois, 469 Computer and Systems Research Lab, 1308 W. Main St., Urbana, IL 61801; e-mail banerjee@crhc.uiuc.edu. More information on Paradigm is available on the Web at URL http://www.crhc.uiuc.edu/Paradigm or ftp://ftp.crhc.uiuc.edu/pub/Paradigm.

Cheri Pancake, formerly Computer’s high-performance computing area editor, coordinated the review of this article and recommended it for publication. Her e-mail address is pancake@cs.orst.edu.

October 1995