appeared in IEEE Computer, Volume 28, Number 10, October 1995

An Overview of 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

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801
Phone: (217) 333-6564
Fax: (217) 333-1910
banerjee@crhc.uiuc.edu

Abstract

Distributed-memory multicomputers such as the the Intel Paragon, the IBM SP-2, and the Thinking Machines CM-5 offer significant advantages over shared-memory multiprocessors in terms of cost and scalability. Unfortunately, extracting all the computational power from these machines requires users to write efficient software for them, which is a laborious process. The PARADIGM compiler project provides an automated means to parallelize programs, written using a serial programming model, for efficient execution on distributed-memory multicomputers. In addition to performing traditional compiler optimizations, PARADIGM is unique in that it addresses many other issues within a unified platform: automatic data distribution, communication optimizations, support for irregular computations, exploitation of functional and data parallelism, and multithreaded execution. This paper describes the techniques used and provides experimental evidence of their effectiveness on the Intel Paragon, the IBM SP-1, and the Thinking Machines CM-5.

Keywords: compilers, distributed memory, multicomputers, parallelizing compilers, parallel processing.

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

† currently working at IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598

(A preliminary version of this paper appeared in the First Int'l Workshop on Parallel Processing, Bangalore, India, December 1994)

1 Introduction

Distributed-memory massively parallel multicomputers can provide the high levels of performance required to solve the Grand Challenge computational science problems. Multicomputers such as the Intel Paragon, the IBM SP-1/SP-2 and the Thinking Machines CM-5 offer significant advantages over shared-memory multiprocessors in terms of cost and scalability. Unfortunately, extracting all the computational power from these machines requires users to write efficient software for them, which is a laborious process. One major reason for this difficulty is the absence of a global address space. As a result, the programmer has to manually distribute computations and data across processors and manage communication explicitly. The PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) project at the University of Illinois addresses this problem by developing an automated means to parallelize and optimize sequential programs for efficient execution on distributed-memory multicomputers.

To understand the complexity of writing programs in a message-passing model, it is useful to examine the parallel code generated by the compiler (which roughly corresponds to what an experienced programmer might write) for a small example. Figure 1a shows an example serial program for Jacobi's iterative method for solving systems of linear equations. In Figure 1b a highly efficient parallel program is shown for an Intel Paragon with a variable number of processors. 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 which are notoriously hard to find. By automating the parallelization process, it will become possible to offer high levels of performance to the scientific computing community at large.

Some of the other major research efforts in this area include Fortran D [1], Fortran 90D [2], and the SUPERB compiler [3]. In order to standardize parallel programming with data distribution directives, High Performance Fortran (HPF) [4] has also been developed in a collaborative effort between researchers in industry and academia. A number of commercial HPF compilers are beginning to appear including products from Applied Parallel Research, Convex, Cray, Digital, IBM, The Portland Group, Silicon Graphics, Thinking Machines and others.

What sets the PARADIGM project apart from other compiler efforts for distributed-memory multicomputers is the broad range of research topics that are being addressed. In addition to per-

```
program jacobi
parameter (np2 = 500, ncycles = 10)
real A(np2, np2), B(np2, np2)
np1 = np2−1
do k = 1, ncycles
    do j = 2, np1
        do i = 2, np1
            A(i, j) = (B(i−1, j) + B(i+1, j)
                + B(i, j−1) + B(i, j+1)) / 4
        end do
    end do
    do j = 2, np1
        do i = 2, np1
            B(i, j) = A(i, j)
        end do
    end do
end do
end
```

(a) Serial version

```
program jacobi
character m$buf(1000)
integer my$p(2), m$bA1, m$bA2, m$bB1, m$bB2
integer m$numdim, m$num(2), m$to(−1:1,−1:1), m$inc
real A(250,250), B(0:251,0:251)
m$numdim = 2     {number of mesh dimensions}
m$num(1) = 2     {minimum mesh configuration}
m$num(2) = 2
call m$getnum(m$num,numnodes())
call m$gridinit(m$numdim,m$num,numnodes(),1)
call m$gridcoord(mynode(),my$p)
m$to(−1,0) = m$gridrel2(my$p,−1,0)
m$to(0,−1) = m$gridrel2(my$p,0,−1)
m$to(0,1) = m$gridrel2(my$p,0,1)
m$to(1,0) = m$gridrel2(my$p,1,0)
m$bA1 = ceil(float(500) / m$num(1))
m$bA2 = ceil(float(500) / m$num(2))
m$bB1 = ceil(float(500) / m$num(1))
m$bB2 = ceil(float(500) / m$num(2))
do k = 1,10
   if (my$p(2) .ge. 1) then
      call csend(0,B(1,1),4*m$bB1,m$to(0,−1),1)
   end if
   if (my$p(2) .le. m$num(2)−2) then
      call crecv(0,B(1,m$bB2+1),4* m$bB1)
   end if
   if (my$p(2) .le. m$num(2)−2) then
      call csend(1,B(1,m$bA2),4*m$bB1,m$to(0,1),1)
   end if
   if (my$p(2) .ge. 1) then
      call crecv(1,B(1,0),4*m$bB1)
   end if
   if (my$p(1) .ge. 1) then
      m$inc = f$pack2(B,4,0,251,0,251,1,1,1,1,m$bB2,1,m$buf)
      call csend(2,m$buf,m$inc,m$to(−1,0),1)
   end if
   if (my$p(1) .le. m$num(1)−2) then
      call crecv(2,m$buf,1000)
      m$inc = f$unpack2(B,4,0,251,0,251,m$bB1+1,m$bA1+1,
                  1,1,m$bB2,1,m$buf)
   end if
   if (my$p(1) .le. m$num(1)−2) then
      m$inc = f$pack2(B,4,0,251,0,251,m$bA1,m$bB1,
                  1,1,m$bB2,1,m$buf)
      call csend(3,m$buf,m$inc,m$to(1,0),1)
   end if
   if (my$p(1) .ge. 1) then
      call crecv(3,m$buf,1000)
      m$inc = f$unpack2(B,4,0,251,0,251,0,0,1,1,m$bB2,1,m$buf)
   end if
   do j = max(2−m$bA2*my$p(2), 1),
            min(499−m$bA2*my$p(2), m$bA2)
      do i = max(2−m$bA1*my$p(1), 1),
               min(499−m$bA1*my$p(1), m$bA1)
         A(i, j) = (B(i−1, j) + B(i+1, j) + B(i, j−1)
                       + B(i, j+1)) / 4
      end do
   end do
   do j = max(2−m$bB2*my$p(2), 1),
            min(499−m$bB2*my$p(2), m$bB2)
      do i = max(2−m$bB1*my$p(1), 1)),
               min(499−m$bB1*my$p(1), m$bB1)
         B(i, j) = A(i, j)
      end do
   end do
end do
end
```

(b) Parallel version for a variable number of processors, with a minimum 2 × 2 mesh configuration

Figure 1: Example program: Jacobi's Iterative Method

forming traditional compiler optimizations to distribute computations and to reduce communication overheads, the research in the PARADIGM project aims at combining the following aspects: (1) performing automatic data distribution for regular computations (2) optimizing communication for regular computations; (3) supporting irregular computations using a combination of compile-time analysis and run-time support; (4) exploiting functional and data parallelism simultaneously; and (5) generating multithreaded message-driven code to hide communication latencies. Current efforts in the project aim at integrating all of these capabilities into the PARADIGM framework. In this article, we briefly describe the techniques used and provide experimental results to demonstrate their utility.

Sidebar - Distributed Memory Compiler Terminology

Data Parallelism - Parallelism that exists by performing similar operations simultaneously across different elements of a data set (**SPMD**, Single Program Multiple Data)

Functional Parallelism - Parallelism that exists by performing potentially different operations 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 run-time analysis)

Data Partitioning - The physical **distribution** of data across the processors of a parallel machine in order 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 just 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 which 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

---

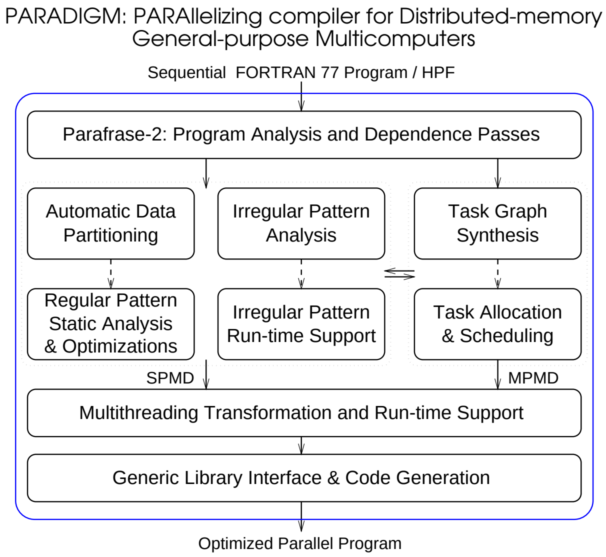


Figure 2: PARADIGM Compiler Overview

2 Compiler Framework

Figure 2 shows a functional illustration of how we envision the complete PARADIGM compilation system. The compiler accepts either a sequential FORTRAN 77 or High Performance Fortran (HPF) program [4] and produces an explicit message passing version (FORTRAN 77 program with calls to the selected message passing library and our run-time system). The following are brief descriptions of the major phases in the parallelization process.

Program Analysis *Parafrase-2* [5] is used as a preprocessing platform to parse the sequential program into an intermediate representation and to analyze the code to generate flow, dependence, and call graphs. Various code transformations, such as constant propagation and induction variable substitution are also performed at this stage.

Automatic Data Partitioning For regular computations, the data distribution is determined automatically by the compiler. It configures the machine into an abstract multidimensional mesh of processors and decides how program data is to be distributed on the mesh [6]. Estimates of the time spent in computation and communication drive the selection of the data distribution. High-level communication operations and other communication optimizations performed in the compiler are reflected in the estimates in order to correctly determine the best distribution.

Regular Computations Using the *owner computes rule*, the compiler divides computation across processors according to the selected data distribution, and generates inter-processor communication for required non-local data. To avoid the overhead of computing ownership at run time, static analysis is used to partition loops at compile time (loop bounds reduction [1]). In addition, a number of optimizations are performed to reduce the overhead of communication [7].

Irregular Computations In many important applications, compile-time analysis is insufficient when communication patterns are data dependent and known only at run time. A subset of these applications has the interesting property that the communication pattern repeats across several steps. PARADIGM approaches these problems through a combination of flexible irregular run-time support and compile-time analysis. Novel features in our approach are the exploitation of spatial locality and the overlapping of computation and communication [8].

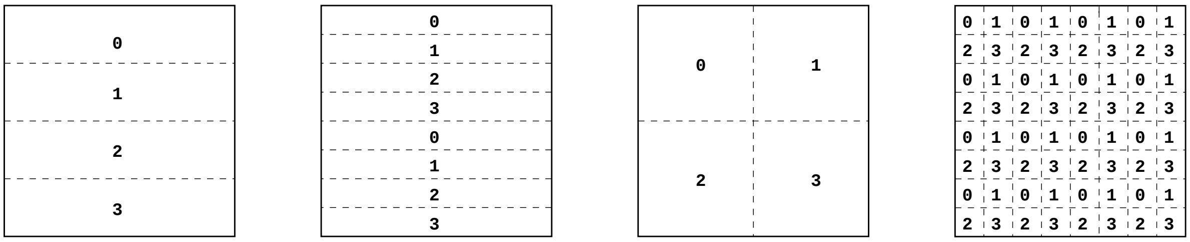
Functional Parallelism Recent research [9] has shown the benefits of simultaneous exploitation of functional and data parallelism for some applications. Such applications can be viewed as a graph composed of a set of data-parallel tasks with precedence relationships which describe the functional parallelism that exists among the tasks. Using this task graph, PARADIGM exploits functional parallelism by determining the number of processors to allocate for each data-parallel task and scheduling them such that the overall execution time is minimized. The techniques used for regular and irregular data-parallel compilation are used to generate code for each of the data-parallel tasks.

Multithreading Message-passing programs normally send messages asynchronously and block when waiting for messages, resulting in lower efficiency. One solution is to run multiple threads on each processor to overlap computation and communication [10]. Multithreading allows one thread to utilize the unused cycles which would otherwise be wasted waiting for messages. Compiler transformations are used to convert message-passing code into a message-driven model, thereby simplifying the multithreading run-time system. Multithreading is most beneficial for programs with a high percentage of idle cycles such that the overhead of switching between threads can be hidden.

Generic Library Interface Support for specific communication libraries is provided through a generic library interface. For each supported library, abstract functions are mapped to corresponding library specific code generators at compile time. Library interfaces have been implemented for Thinking Machines CMMD, Parasoft Express, MPI, Intel NX, PVM, and PICL. Execution tracing, as well as support for multiple platforms, is also provided in Express, PVM, and PICL. The portability of this interface allows the compiler to easily generate code for a wide variety of machines.

In the remainder of this paper, each of the major areas within the compiler will be described in more detail. Section 3 outlines the techniques used in automatic data partitioning. Section 4 describes the analysis and optimizations performed for regular computations, while Section 5 describes our approach for irregular computations. Section 6 explores the simultaneous use of functional and data parallelism, and Section 7 reports on multithreading message-driven code.

Sidebar - Data Distribution



(a) `block , *` (b) `cyclic(`k`) , *` (c) `block , block` (d) `cyclic(`k_1`) , cyclic(`k_2`)`

Examples of data distributions for a two-dimensional array

Arrays are physically distributed across processors to efficiently use available memory and improve the locality of reference in parallel programs. In High Performance Fortran [4], either the programmer or the compiler must specify the distribution of program data. Several examples of data distributions are shown 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($\frac{N}{P}$). A cyclic distribution is simply a block-cyclic distribution with a block size of one, cyclic(1).

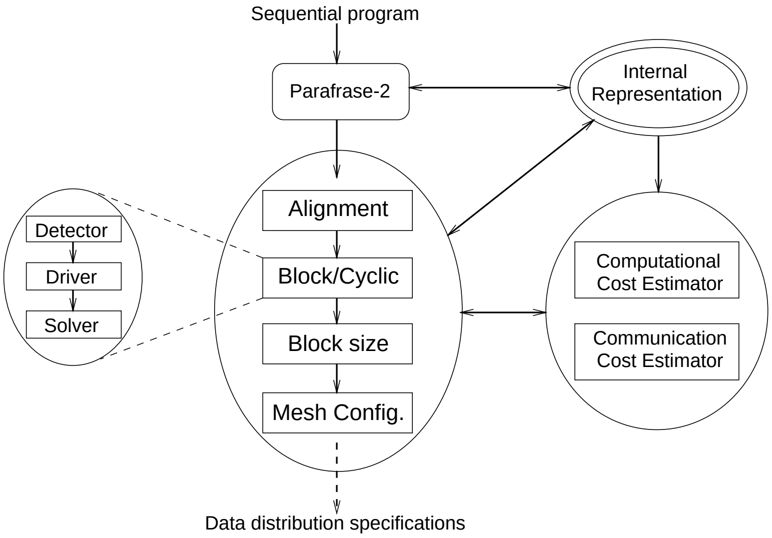


Figure 3: Automatic Data Partitioning Overview

3 Automatic Data Partitioning

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

- reduces the burden on the programmer
- improves program portability and machine independence
- improves the selection of data distributions

In the compiler, data partitioning decisions are made in a number of distinct phases (as illustrated in Figure 3) [6]. Often, there is a tradeoff between minimizing interprocessor communication and exploiting all available parallelism; the communication and the computational costs imposed by the underlying architecture must both be taken into account. These costs are generated using architectural parameters for each target machine. With the exception of the architecture-specific costs, the partitioning algorithm is machine independent.

Below are brief descriptions of each of the major phases performed during the partitioning pass. Each phase involves identification of data distribution preferences by a `detector` module, assignment of costs by a `driver` to quantify the estimated performance impact of those preferences, and the resolution of any conflicts by a `solver`.

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 be between different pairings of dimensions (inter-dimensional alignment) as well as by an offset or stride within a given pair of dimensions (intra-dimensional alignment). Currently, only inter-dimensional alignment is performed in the partitioning pass.

Block/Cyclic Distribution Once array alignment has been performed, the distribution pass determines whether each array dimension should be distributed in a blocked or cyclic manner. Array dimensions are first classified by their communication requirements. If the communication in a mesh dimension is recognized as a nearest-neighbor pattern, it indicates the need for a blocked distribution. For dimensions that are only partially traversed (less than a certain threshold), a cyclic distribution may be more desirable for load balancing. Using alignment information from the previous phase, the array dimensions that cross-reference each other are assigned the same kind of partitioning to ensure the intended alignment.

Block Size Selection When a cyclic distribution is chosen, the compiler is able to make further adjustments on the block size giving rise to block-cyclic partitionings. Since only a cyclic distribution is chosen in the previous phase, in order to improve the load balance for partially traversed array dimensions, a closer examination of the communication costs must be performed. 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 After all the distribution parameters have been determined, the cost estimates are functions of only the number of processors in each mesh dimension. For each set of aligned array dimensions, the compiler determines if there are any parallel operations performed. If no parallelism exists in a given dimension, it is collapsed onto a single processor. If there is only one dimension that has not been collapsed, all processors are assigned to this dimension. In the case of 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 may also desire to generate a HPF program containing the directives which specify the selected partitioning. This technique allows the partitioning pass to be used as an independent tool while remaining integrated with the compilation system. Furthermore, being integrated ensures that the partitioning pass is always aware of the optimizations performed by the compiler. For more complex programs, it is also possible to further improve performance by redistributing data at selected points in the program. The static partitioner is currently being extended to automatically determine when such dynamic data partitionings are useful.

4 Regular Computations

For regular computations in which the communication pattern can be determined at compile time, PARADIGM uses static analysis to partition computation across processors and to generate optimized inter-processor communication. To do this analysis efficiently, the compiler needs a mechanism to describe partitioned data and iteration sets. Processor Tagged Descriptors (PTDs) are used to provide a uniform representation of these sets for every processor [11]. Operations on PTDs are extremely efficient, simultaneously capturing the effect on all processors in a given dimension.

PTDs, however, are not general enough to handle the most complicated array distributions, references, and loop bounds that are occasionally found in real codes (see Figure 4). For these complex cases, 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 [12]. To implement Fourier-Motzkin projection, Mathematica™ (a powerful off-the-shelf symbolic analysis system) is linked with the compiler to provide a high level of symbolic support as well as 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 is able to compile a larger proportion of programs without jeopardizing compilation speed.

```
         REAL A(170)
         REAL B(120,120)
!HPF$    PROCESSORS MESH(4,2)
!HPF$    TEMPLATE T(170,2)
!HPF$    ALIGN A(k) WITH T(k,1)
!HPF$    DISTRIBUTE T(CYCLIC(5), BLOCK)      ONTO MESH
!HPF$    DISTRIBUTE B(CYCLIC(3), CYCLIC(7))ONTO MESH
         DO i = 3,40
           DO j = 2,i-1
             A(4*i+5) = B(2*i+j-1,3*i-2*j+1)
           END DO
         END DO
```

Figure 4: Example loop with complex array references and distributions

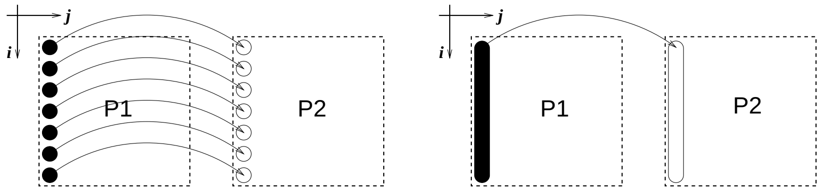
The performance of the resulting parallel program also greatly depends on how well its interprocessor communication has been optimized. Since the start-up cost of communication (*overhead*) tends to be several orders of magnitude greater than either the per-element computation cost or the per-byte transmission cost (*rate*), frequent communication can easily dominate the execution time. A linear point-to-point transfer cost of a message of m bytes is used as a basis for the communication model:

$$\mathit{transfer}(m) = \mathit{overhead} + \mathit{rate} \times m$$

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

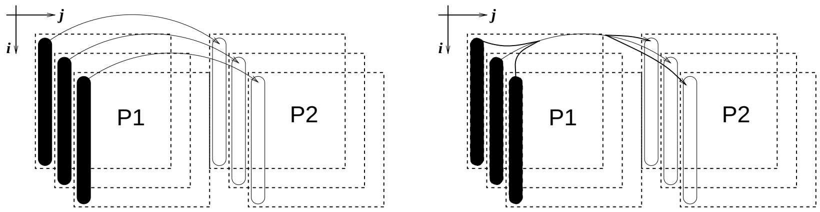
Several optimizations are employed that combine messages in different ways to amortize the start-up cost and thereby reducing the total amount of communication overhead in the program [1, 7]. In loops where there are no cross-iteration dependencies, parallelism is extracted by independently executing groups of iterations on separate processors. For independent references, *message coalescing*, *message vectorization*, and *message aggregation* are used to reduce the overhead associated with frequent communication. For references within loops that contain cross-iteration dependencies, *coarse-grain pipelining* is employed to optimize 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



Before After

(a) Message Vectorization



Before After

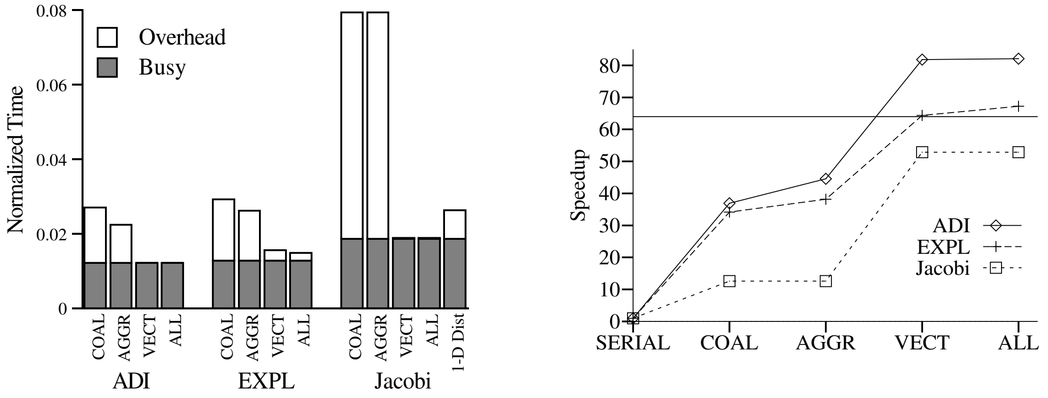
(b) Message Aggregation

Figure 5: Optimizations used to reduce overhead associated with frequent communication

references, redundant communication is detected and coalesced into a single message, allowing the data to be reused rather than communicated for every reference. For different sections of a given array, individual elements are coalesced by performing a union of 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 Non-local elements of an array that are indexed within a loop nest can also be *vectorized* into a single larger message instead of being communicated individually (see Figure 5a). Dependence analysis is used to determine the outermost loop at which the combining can be applied. The element-wise messages are combined, or *vectorized*, as they are lifted out of the enclosing loop nests to the selected vectorization level. Vectorization reduces the total number of communication operations (hence the total overhead) while increasing the message length.

Message Aggregation Multiple messages to be 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



SERIAL	= original unmodified serial code	VECT	= COAL + message vectorization
COAL	= statically partitioned parallel program with message coalescing	AGGR	= COAL + message aggregation
		ALL	= COAL + message vectorization and aggregation

Figure 6: Comparison of Message Coalescing, Vectorization, and Aggregation (64 processor CM-5)

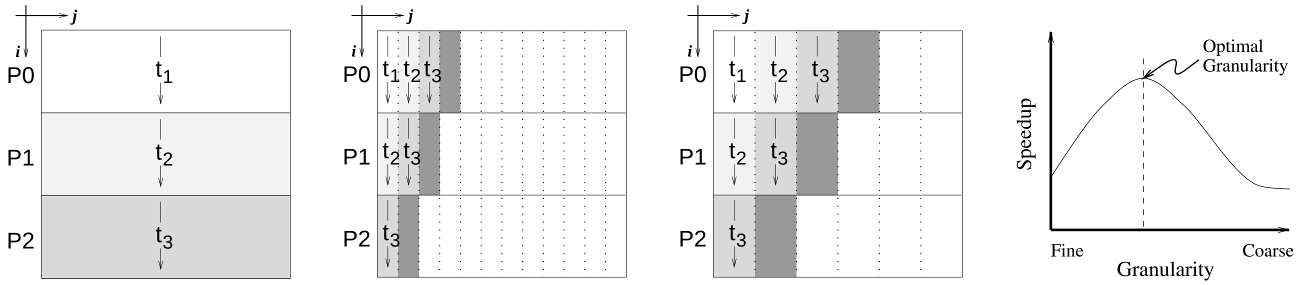
pairs are then combined into a single communication operation (see Figure 5b). Aggregation can be performed on communication operations of individual data references as well as vectorized communication operations. The gain from aggregation is similar to vectorization in that the total overhead is reduced at the cost of increasing the message length.

To illustrate the efficacy of these optimizations, the performance of several program fragments executed on a 64 processor CM-5 is shown in Figure 6. The automatic data distribution pass selected linear (1-D) partitionings for both ADI Integration (Livermore kernel 8 with 16,000-element arrays) and Explicit Hydrodynamics (Livermore kernel 18 with 16,000-element arrays), and a 2-D partitioning for Jacobi's iterative method (similar to that previously shown in Figure 1 with 1000×1000 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

The relative effectiveness of each optimization can be seen by examining the amount of overhead eliminated as the optimizations are incrementally applied.



(a) Sequential (b) Fine Grain (c) Coarse Grain

Figure 7: Pipelined Execution of Recurrences

It is also interesting to notice that an additional run of a 1-D partitioned version of Jacobi shows a higher overhead compared to the compiler-selected 2-D version. This shows the effectiveness of the automatic data partitioning pass since it was able to select the best distribution despite the small differences in performance. For larger machine sizes and more complex programs, the utility of automatic data distribution will be even more apparent as the communication costs become greater for inferior data distributions.

Coarse Grain Pipelining In cases where there are cross-iteration dependencies due to recurrences, it is not possible to immediately execute every iteration in parallel. Often, however, there is the opportunity to overlap parts of the loop execution, synchronizing to ensure that the data dependencies are enforced.

To illustrate this technique, assume an array is block partitioned by rows, and dependencies exist from the previous row and previous column. In Figure 7a, each processor performs an operation on every element of the rows it owns before sending the border row to the waiting processor, thereby serializing execution of the entire computation. Instead, in Figure 7b, the first processor can compute the elements of one partitioned column and then send the border element of that column to the next processor such that it can begin its computation immediately. Ideally, if communication has zero overhead, this is the most efficient form of computation, since no processor will wait unnecessarily. However, as discussed earlier, the cost of performing numerous single-element communications can be quite expensive compared to the small strips of computation. To address this problem, this overhead can be reduced by increasing the granularity of the communication (see Figure 7c). An analytic pipeline model has been developed using estimates of computation

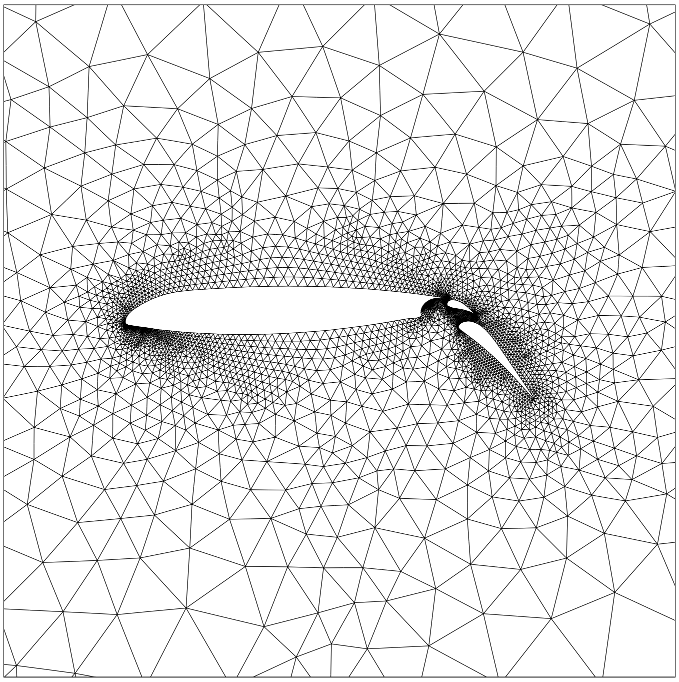


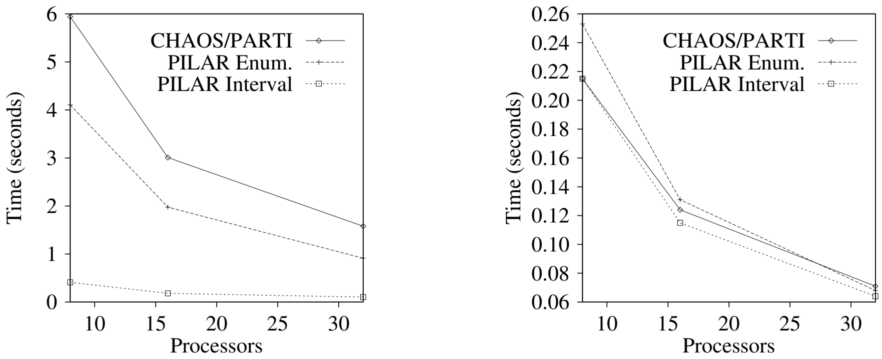
Figure 8: Finite Element Airfoil Grid

and communication to allow the compiler to automatically select a granularity that results in near-optimal performance [7].

5 Irregular Computations

In many important applications compile-time analysis is insufficient when the required communication patterns are data dependent and thus are only known at run time. For example, the computation of airflow and surface stress over an airfoil may utilize an irregular finite element grid, such as the one shown in Figure 8 (which contains 4,720 vertices and 13,722 edges). To efficiently run such irregular applications on a massively parallel multicomputer, run-time compilation techniques can be used [13]. The dependency structure of the program is analyzed in a preprocessing step before the actual computation occurs. If the same structure of a computation is maintained across several steps, this preprocessing can be reused, amortizing its 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



(a) Schedule (inspector)

(b) Redistribute (executor)

Figure 9: Edge redistribution for Rotor on the IBM SP-1

translated into local indices. During the executor phase, elements are communicated based on this preprocessing analysis. In order to simplify the implementation of inspectors and executors, irregular run-time 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 [13]. The internal representation of communication patterns in such systems is somewhat restricted; they represent irregular patterns that are completely enumerated or regular block patterns. Neither optimizes regular and irregular accesses together nor efficiently supports the small regular blocks that arise in irregular applications written for the exploitation of spatial cache locality. Moreover, systems such as CHAOS/PARTI do not provide non-blocking communication primitives which can further increase performance.

All of these problems are addressed in the Parallel Irregular Library with Application of Regularity (PILAR) [8], PARADIGM's IRTS for irregular computations. PILAR is written in C++ to easily support different internal representations of communication patterns; allowing for efficient handling of a wide range of applications, from fully irregular to regular, using a common framework. PILAR uses intervals for the description of small regular blocks (as discussed earlier), and enumeration for patterns with little or no regularity. The object-oriented nature of the library simplifies the implementation of new representations as well as the interactions among objects that have different internal representations.

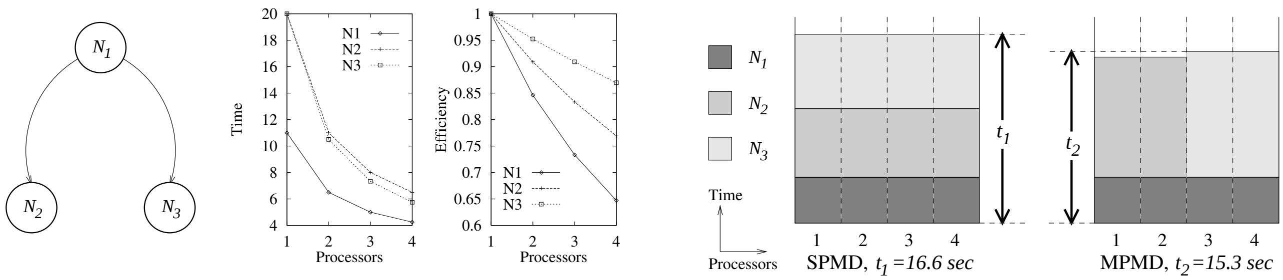
An experiment is performed to evaluate the effectiveness of PILAR in exploiting spatial regularity in irregular applications. The overhead of redistributing the edges of an unstructured grid is measured after a partitioner has assigned nodes to processors. We assume a typical CSR (Compressed Sparse Row) or Harwell-Boeing initial layout in which edges of a given grid node are contiguous in memory. Redistribution is done in two phases: the first phase (inspector) computes a schedule that captures the redistribution of the edges and sorts the new global indices; the second phase (executor) redistributes the array with the previously computed schedule using a global data exchange primitive.

The experiment uses a large unstructured grid, *Rotor*, from NASA. A large ratio (9.40) between the maximum degree and the average degree of a node in this grid would cause a two-dimensional matrix representation of the edges to be very inefficient. Multidimensional optimizations in CHAOS or PILAR (with enumeration) cannot be used. The performance of CHAOS/PARTI is 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 9, which clearly shows the benefit of using the more compact interval representation. Further experiments also show that only three edges per grid node are required to benefit from an interval based representation in the SP-1 [8].

Even with adequate IRTS, the generation of efficient inspector/executor code for irregular applications is fairly complex. In PARADIGM, compiler analysis for irregular computations will be used to detect reuse of preprocessing, insert communication primitives, and highlight opportunities to exploit spatial locality. After performing this analysis, the compiler will generate inspector/executor code with embedded calls to PILAR routines.

6 Functional and Data Parallelism

The efficiency of data-parallel execution tends to drop off for larger numbers of processors (for a given problem size) or for smaller problem sizes (for a given number of processors). By exploiting functional parallelism in addition to data parallelism, the overall execution efficiency of a program can sometimes be improved. A task graph, known as a Macro Dataflow Graph (MDG), is used to represent both the functional and data parallelism available in a program. The MDG for a given program is a weighted directed acyclic graph (DAG) with nodes representing data-parallel routines in the program and edges representing precedence constraints among these routines. In the MDG, data parallelism is implicit in the weight functions of the nodes while functional parallelism is captured by the precedence constraints among nodes.



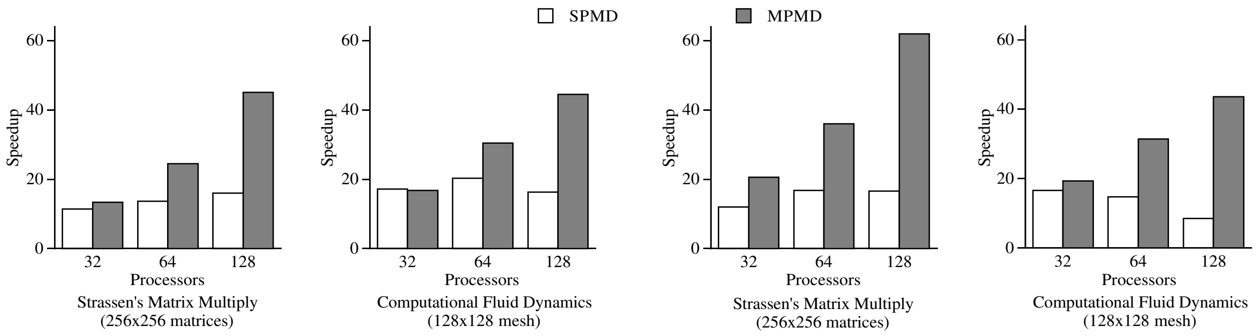
(a) Macro Dataflow Graph (b) Allocation and Scheduling

Figure 10: Example of Functional Parallelism

The weights of the nodes and edges are based on the processing and data redistribution costs. The processing cost is the computation and communication time required for the execution of 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 pair of routines. The time required for this data redistribution depends on the number of processors as well as the data distributions used by the routines.

To determine the best execution strategy for a given program, an allocation and scheduling approach is used on the MDG. Allocation determines the number of processors to use for each node, while scheduling results in an execution scheme for the allocated nodes on the target multicomputer. Figure 10a shows an MDG with three nodes (N_1, N_2, and N_3) along with the processing costs and efficiencies of the nodes as a function of the number of processors they use. For this example, assume there are no data redistribution costs among the three routines. Given a four processor system, two schemes of execution for the program are shown pictorially in Figure 10b. The first scheme exploits pure data parallelism with all routines using four processors. The second scheme exploits both functional and data parallelism with routines N_2 and N_3 executing concurrently and using two processors each. As shown by this example, good allocation and scheduling can decrease the program execution time.

The allocation and scheduling algorithms in PARADIGM are based on the mathematical forms of the processing and data redistribution cost functions; it can be shown that they belong to a class of functions known as *posynomials*. This property is used to formulate the problem with a form of convex programming for optimal allocation. After allocation, a list scheduling policy is used for scheduling the nodes on a given system. The finish time obtained using this scheme has been



(a) CM-5

(b) Paragon

Figure 11: SPMD/MPMD Performance Comparison

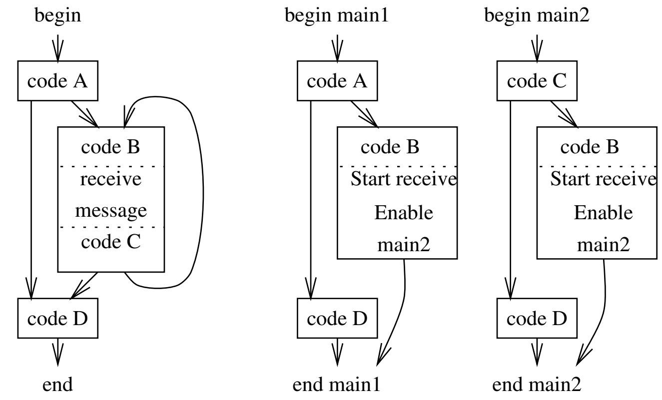
shown to be within a factor of the optimal finish time in theory; in practice, this factor is small [9].

In Figure 11, the performance of the allocation and scheduling approach is compared to that of a pure data-parallel approach. Speedups are computed for the PARAGON and CM-5 for a pair of applications. Performance using the allocation and scheduling approach is identified as "MPMD," and performance for the pure data-parallel scheme is "SPMD." 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 a large number of processors, the performance of the MPMD execution relative to SPMD is improved by a factor of about two to three. These results demonstrate the utility of the allocation and scheduling approach.

7 Multithreading

When the resulting parallel program has a high percentage of idle cycles, multithreading can be used to further improve performance. By running multiple threads on each processor, one of the threads can utilize the cycles which would otherwise be wasted waiting for messages. To support multithreaded execution, message passing code is first generated by the compiler for a number of virtual processors greater than the number of physical processors in a given machine. Multiple virtual processors are then mapped onto physical processors, resulting in multiple threads of execution for each physical processor.

In order to execute multithreaded code efficiently, compiler transformations are used to convert message-passing code into a message-driven model, thereby simplifying the multithreading run-time



(a) Original
Control Flow Graph

(b) Transformed
Control Flow Graph

Figure 12: Transformation of the While Statement

system (MRTS).[1] The transformation required is simple for code without conditionals but becomes more complex when conditionals and loops are included. Although this section only presents the transformation for converting *while* loops to message-driven code, similar transformations can be performed on other control structures [10].

The transformation of the *while* loop is shown in Figure 12. Figure 12a shows the control flow graph of a message-passing program containing a receive in a *while* loop. Figure 12b shows the transformed code. In Figure 12b, *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 and the thread ends. 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, and if it is true, *main2* enables another invocation of itself and returns to the MRTS. Otherwise, code D is executed and the thread ends. Multiple copies of *main1* and *main2* can be executed on the processors to increase the degree of multithreading.

This transformation was performed on the following four scientific applications written in the SPMD programming model with blocking receives:

[1] In the message-driven model, receive operations are used to switch between threads and therefore must return control to the multithreading run-time system.

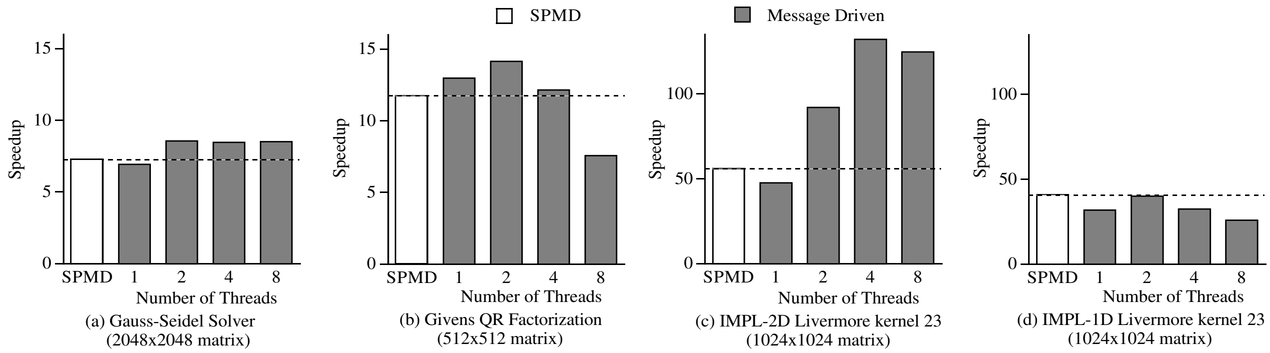


Figure 13: Speedup of Message Driven Threads (64 processor CM-5)

GS	-	Gauss-Seidel Iterative Solver
QR	-	Givens QR factorization of a dense matrix
IMPL-2D	-	2-D distribution of Implicit Hydrodynamics
IMPL-1D	-	1-D distribution of Implicit Hydrodynamics

All were run with large matrices on the CM-5.

Figure 13 compares the speedup of SPMD code with that of message-driven code (with varying numbers of threads per processor). For the Gauss Seidel and Givens QR applications, the amount of available parallelism inhibits the speedup. On the other hand, cache effects produce super-linear speedup for Implicit Hydrodynamics. The message-driven versions of the code outperform the SPMD versions in all cases except IMPL-1D, where multithreading causes a significant increase in communication costs. For the other applications improvement is seen when 2 to 4 threads are used, but the exact number of threads that produces the maximum speedup varies. This indicates the number of threads required for optimal speedup is somewhat application dependent.

8 Conclusions

PARADIGM is a flexible parallelizing compiler for multicomputers. It can automatically distribute program data and perform a variety of communication optimizations for regular computations as well as provide support for irregular computations using compilation and run-time techniques. For programs which have 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. We believe that all these methods are useful for compiling a wide range of applications for distributed-memory multicomputers.

References

[1] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD Distributed Memory Machines," *Communications of the ACM*, vol. 35, pp. 66–80, Aug. 1992.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka, "Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers, Design, Implementation, and Performance Results," in *Proceedings of the 7th ACM International Conference on Supercomputing*, (Tokyo, Japan), pp. 351–360, July 1993.

[3] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, vol. 1, pp. 31–50, Aug. 1992.

[4] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, *The High Performance Fortran Handbook*. Cambridge, MA: The MIT Press, 1994.

[5] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors," in *Proceedings of the 18th International Conference on Parallel Processing*, (St. Charles, IL), pp. II:39–48, Aug. 1989.

[6] M. Gupta and P. Banerjee, "PARADIGM: A Compiler for Automated Data Partitioning on Multicomputers," in *Proceedings of the 7th ACM International Conference on Supercomputing*, (Tokyo, Japan), July 1993.

[7] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee, "Compiler Optimizations for Distributed Memory Multicomputers used in the PARADIGM Compiler," in *Proceedings of the 23rd International Conference on Parallel Processing*, (St. Charles, IL), pp. II:1–10, Aug. 1994.

[8] A. Lain and P. Banerjee, "Exploiting Spatial Regularity in Irregular Iterative Applications," in *Proceedings of the 9th International Parallel Processing Symposium*, (Santa Barbara, CA), pp. 820–827, Apr. 1995.

[9] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," in *Proceedings of the 23rd International Conference on Parallel Processing*, (St. Charles, IL), pp. II:116–125, Aug. 1994.

[10] J. G. Holm, A. Lain, and P. Banerjee, "Compilation of Scientific Programs into Multithreaded and Message Driven Computation," in *Proceedings of the 1994 Scalable High Performance Computing Conference*, (Knoxville, TN), pp. 518–525, May 1994.

[11] E. Su, D. J. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers," in *Proceedings of the 1994 International Conference on Parallel Architectures and Compilation Techniques*, (Montréal, Canada), pp. 123–132, Aug. 1994.

[12] C. Ancourt and F. Irigoin, "Scanning Polyhedra with DO Loops," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, (Williamsburg, VA), pp. 39–50, Apr. 1991.

[13] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse," in *Proceedings of Supercomputing '93*, (Portland, OR), pp. 361–370, Nov. 1993.