The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers

Prithviraj Banerjee John A. Chandy Manish Gupta† 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, U.S.A.
banerjee@crhc.uiuc.edu

Abstract – The PARADIGM *compiler project provides an automated means to parallelize programs, written in 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, synthesis of high-level communication, communication optimizations, irregular computations, functional and data parallelism, and multithreaded execution. This paper describes the techniques used and provides experimental evidence of their effectiveness.*

1 Introduction

Distributed-memory massively parallel multicomputers can provide the high levels of performance required to solve the Grand Challenge computational science problems [16]. Distributed-memory multicomputers such as the Intel iPSC/860, the Intel Paragon, the IBM SP-1 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 distribute code and data across processors and manage communication between tasks explicitly.

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 and a Fulbright/MEC fellowship.

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

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. Some of the other major research efforts in this area include Fortran D [9], Fortran 90D [2], the SUIF compiler [1], and the SUPERB compiler [3]. In addition to performing traditional compiler optimizations to distribute computations and to reduce communication overheads, PARADIGM is unique in its ability to: (1) perform automatic data distribution for regular computations, which are conventionally specified by the user through directives; (2) generate high-level communication primitives; (3) optimize communication for regular computations; (4) support irregular computations using a combination of compile-time analysis and run-time support; (5) exploit functional and data parallelism simultaneously; and (6) generate multithreaded message-driven code to tolerate communication latencies. Current efforts in the project aim at integrating all of these capabilities into the PARADIGM framework. In this paper, we describe the techniques used and provide experimental results to demonstrate their feasibility.

2 Compiler Framework

Figure 1 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) [8] program and produces a parallel program optimized for a target machine. Following are brief descriptions of the major areas in the parallelization process:

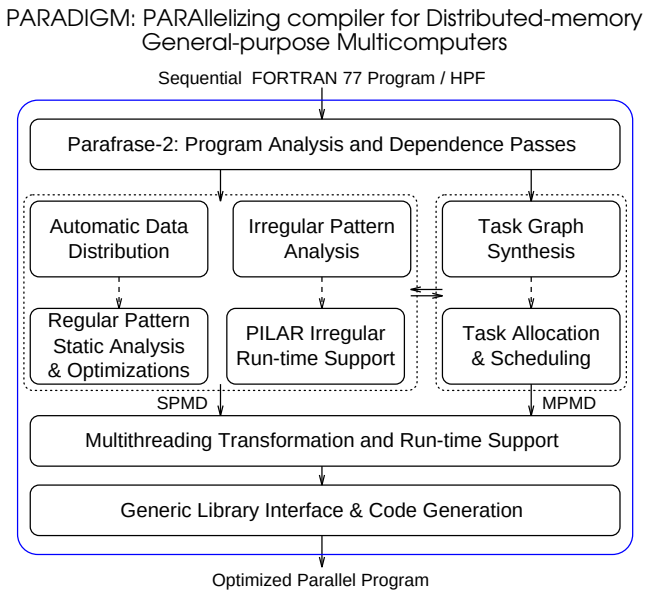


Figure 1: PARADIGM Compiler Overview

Program Analysis *Parafrase-2* [18] 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 of arrays is determined automatically by the compiler. A constraint-based algorithm selects an abstract multidimensional mesh topology along with how program data are to be distributed on the mesh [7]. Constraints are based on execution time estimates that are determined using parameterized cost models of computation and communication. High-level communication operations and other communication optimizations performed in the compiler are reflected in the cost models in order to correctly determine the best distribution.

Regular Computations Computation is divided among processors using the *owner computes* rule which states that all computations for a data element are performed by the processor which owns that element. To avoid the overhead of computing ownership at run time, static optimizations partition loops (loop bounds reduction [9]) such that each processor executes only those iterations that write to its local memory [22].

To describe partitioned data and iteration sets, Processor Tagged Descriptors (PTDs) are used to provide a uniform representation of the partitioning for every processor [23]. Operations on PTDs are extremely efficient, simultaneously capturing the effect on all processors in a given dimension.

Array references are also analyzed to detect the need for communication. PTDs are constructed to describe iterations requiring non-local data, the processors involved, and the exact regions of the arrays to be sent or received. Once the communication descriptors have been computed for individual references, various communication optimizations can be performed [10, 17, 22].

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 unknown communication pattern repeats across several iterations. PARADIGM approaches these problems through a combination of flexible irregular run-time support [13] and compile-time analysis. The main differences with previous work on run-time compilation techniques [19, 21] are the exploitation of spatial locality and the overlapping of computation and communication [12].

Functional Parallelism Recent research efforts [4, 20, 24] show the benefits of simultaneous exploitation of functional and data parallelism for many applications. We define *functional parallelism* as the parallelism that exists between various routines of a given program, and *data parallelism* as the parallelism that exists within a given routine. After extracting a task graph from the input program, PARADIGM exploits functional parallelism by determining the number of processors to allocate for each routine such that the overall execution time is minimized.

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 [11]. 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.

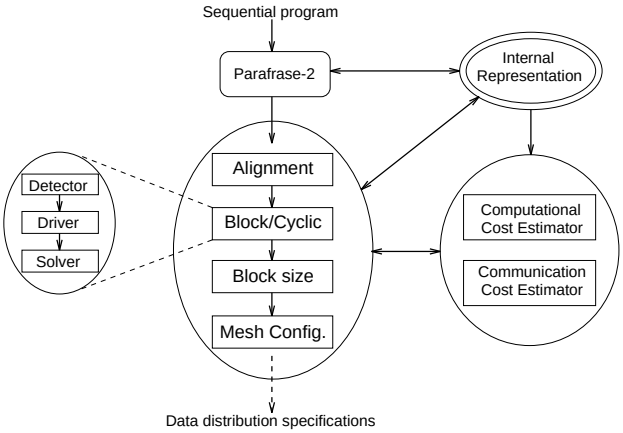
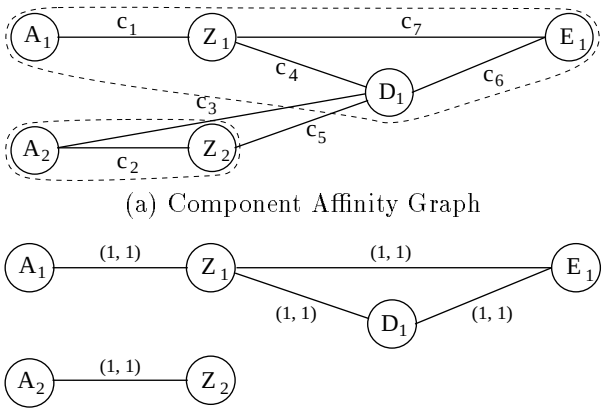


Figure 2: Automatic Data Partitioning Overview



(a) Component Affinity Graph

(b) Block-size Constraint Graph

Figure 3: Partitioning Example (`TRED2`)

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 [15], the Intel NX communication library, PVM [5], and PICL [6]. 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 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 various communication optimizations performed for regular computations, while Section 5 describes our approach for irregular computations. Section 6 explores the simultaneous application of functional and data parallelism and, Section 7 reports on multithreading message-driven code.

3 Automatic Data Partitioning

Performing automatic data partitioning provides several advantages: it reduces the burden on the programmer, improves program portability and machine-independence, and improves the selection of data distributions with a greater knowledge of the optimizations performed in the compiler [7]. Data partitioning decisions are made in a number of distinct phases. The internal structure of the automatic data partitioning module is illustrated in Figure 2.

A constraint-based approach is employed in each phase to determine the partitioning from the characteristics of the program. Constraints are based on execution time estimates determined using parameterized models of computation and communication costs. 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.

Array Alignment The alignment pass identifies the constraints on alignment among various array dimensions and groups them into classes that are to be mapped to the same mesh dimension. For this problem, we extend the Component Affinity Graph (CAG) framework introduced by Li and Chen [14] with arbitrary weights.

The CAG of a program is now a weighted graph that has nodes representing array dimensions, and edges representing constraints on alignment between dimensions. Edge weights represent the importance of honoring the alignment relationships. An example CAG from `TRED2` (a routine from the Eispack library) is shown in Figure 3a. The component alignment problem involves partitioning the node set of a CAG into disjoint subsets such that the total weight of edges cut is minimized. Our main contribution to this specific problem is in the method of assigning weights to the edges such that these weights reflect the communication costs saved by aligning the corresponding array dimensions.

Block/Cyclic Distribution This pass determines, for each array dimension, whether it should be distributed in a blocked or cyclic manner. Note that blocked and cyclic distributions are two extremes of a general distribution commonly referred to as block-cyclic. 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. A cyclic distribution is simply a block-cyclic distribution with a block size of one.

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. Quality measures are constructed for a general block-cyclic distribution and are evaluated for the block sizes of the two cases. Using alignment information for the previous phase, the array dimensions that cross-reference each other are assigned the same kind of partitioning based on their quality measures 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. The block-size constraints and their quality measures are represented in the Block-size Constraint Graph (BCG). The BCG can be thought of as a CAG after alignment has been performed. Edges now represent the block-size constraints between the aligned dimensions (see Figure 3b). Each edge contains an estimate of the extra communication cost incurred if the constraint is violated. In the absence of any block-size constraint, this value is set to one. In general, the BCG may have cycles that can introduce conflicts which are resolved by the compiler through the use of cost estimates [7].

Mesh Configuration After all of the distribution parameters have been determined, the cost estimates are only functions of 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 in the program. If no parallelism exists in a given dimension, it is collapsed into 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 for the expected execution time for each configuration.

At this point, the distribution information is passed on in the compiler to direct the remainder of the parallelization process. The user may also desire to generate a HPF program containing the directives which specify the selected partitioning. If the resulting HPF program is recompiled later, the partitioning process can be bypassed using the distribution directives instead. This technique allows the partitioning pass to be used as an independent tool while remaining integrated with the compilation system. This ensures that the partitioning pass is always aware of the optimizations performed by the compiler.

For further information, an evaluation of the static data partitioning technique has been presented in previous work [7, 17]. For more complex programs, performance can be further improved by redistributing data at selected points in the program. The static partitioner is currently being extended to determine when such dynamic data partitionings are useful.

4 Optimizing Regular Computations

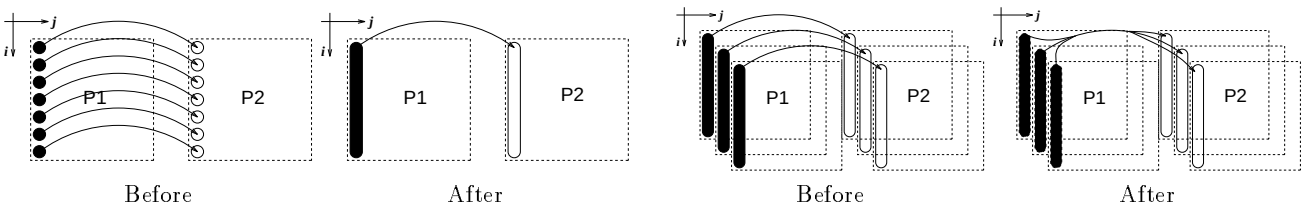
For regular computations, in which the data to be communicated can be determined at compile-time, several optimizations can be performed. Since the start-up cost of communication tends to be much greater than either the per-element computation cost (iPSC/860 $\approx 6000\times$, CM-5 $\approx 2500\times$) or the per-byte transmission cost (iPSC/860 $\approx 400\times$, CM-5 $\approx 700\times$), frequent communication will easily dominate the execution time. By examining the point-to-point transfer cost of a message of m bytes,

$$transfer(m) \;=\; ovhd + rate \times m$$

it can be seen that if the communication overhead is fixed for a given destination, the overhead of communication can be amortized by combining messages.

Several optimizations [10, 17] are employed that combine messages in several ways to reduce the total amount of communication overhead. 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 Separate communication for different references to the same data is unnecessary if the data has not been modified between uses. When statically analyzing access patterns, redundant communications are detected and coalesced into a single message, allowing the data to be reused rather than communicated for every reference. For sections of arrays, unions of overlapping PTD index sets ensure that each unmodified data element is communicated



(a) Message Vectorization (b) Message Aggregation

Figure 4: Optimizations used to reduce overhead associated with frequent communication

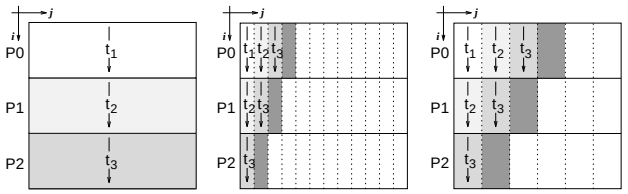
only once. Coalescing is always beneficial since entire communication operations are eliminated.

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 4a). Dependence analysis is used to determine the outermost loop at which the combining can be applied. The "itemwise" 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 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 pairs are then combined into a single communication operation (see Figure 4b). The gain from aggregation is similar to vectorization in that multiple communication operations can be eliminated at the cost of increasing the message length. Aggregation can be performed on communication operations of individual data references as well as vectorized communication operations.

Coarse Grain Pipelining In loops where there are no cross-iteration dependencies, parallelism is extracted by independently executing groups of iterations on separate processors. However, in cases where there are cross-iteration dependencies due to recurrences, it is not possible to immediately execute every iteration. Often, 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 5a, 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 5b, the first processor can compute one partitioned column of elements and then send the border element of that column the next processor such that it can begin computation immediately. Ideally, if communication has zero overhead, this is the most efficient form of computation, since no processor will wait unnecessarily. However, 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 allowing the compiler to automatically control the granularity of the communication [17] (see Figure 5c).



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

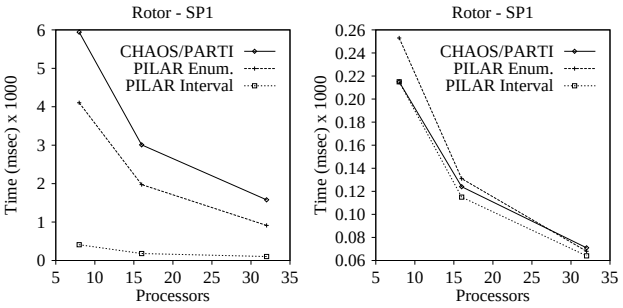
Figure 5: Pipelined Execution of Recurrences

Evaluation of these optimizations has been performed demonstrating their efficacy for regular computations [10, 17].

5 Optimizing Irregular Computations

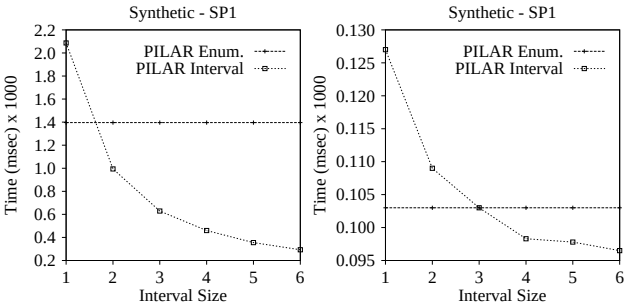
To efficiently run irregular applications on a massively parallel multicomputer, run-time compilation techniques [21] can be used. The dependency structure of the program is analyzed before the actual computation in a preprocessing step. If the same structure of a computation is maintained across several iterations, this preprocessing can be reused, amortizing its cost. In practice, this concept is implemented with two sequences of code: an inspector for the preprocessing and an executor for 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



(a) Schedule (b) Redistribute

Figure 6: Edge redistribution on the IBM SP-1



(a) Schedule (b) Redistribute

Figure 7: Granularity of PILAR on the IBM SP-1

indices. During the execution of computation, data are exchanged 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 [19]. A previous study [12] justified the need for non-blocking communication exchange primitives not available in CHAOS/PARTI. Moreover, the internal representation of communication patterns in such systems is somewhat restricted. These approaches represent irregular patterns that are either completely enumerated or regular block patterns. Neither optimizes regular and irregular accesses together nor efficiently supports spatial regularity[1] in irregular applications.

All these problems are addressed in the Parallel Irregular Library with Application of Regularity (PILAR), PARADIGM's IRTS for irregular computations. PILAR is a C++ library that can support different internal representations of communication patterns. This allows efficient handling of a wide range of applications, from fully irregular to regular, using a common framework. An interval-based representation is used to exploit spatial regularity, and enumeration is used for patterns with little regularity. The object-oriented nature of the library simplifies both the implementation of new representations and the interaction between objects that have different internal representations.

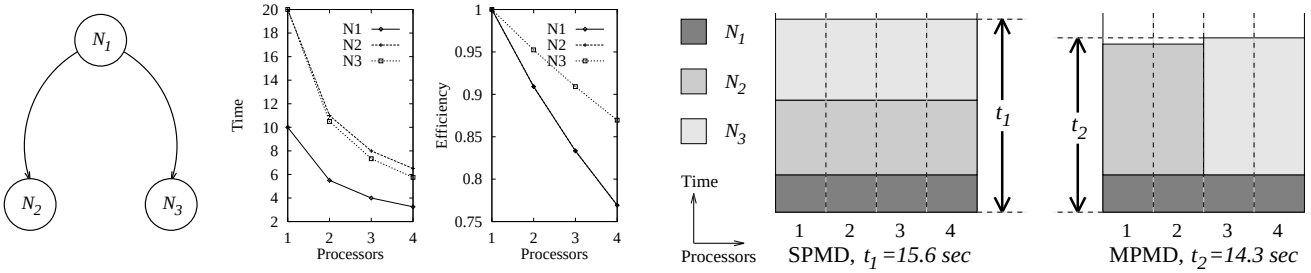
Two experiments are performed to evaluate the effectiveness of PILAR in exploiting spatial regularity in irregular applications. The overhead of redistributing the edges of an unstructured grid is measured after a partitioner has assigned nodes to processors. We assume a typical CSR (Compressed Sparse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

The first experiment uses a large unstructured grid, *Rotor*, from NASA (similar results with other grids are shown in [13]). A large ratio (9.40) between the maximum degree and the average degree 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 both PILAR with enumeration and intervals during the two phases of the redistribution. Results for a 32-processor IBM SP-1 appear in Figure 6, which clearly shows the benefit of using the more compact interval representation.

The second experiment shows the threshold of regularity at which it is advantageous to use intervals in PILAR instead of enumeration. A synthetic, constant-degree grid is used, and the number of intervals that represent the edges associated with a node is arbitrarily varied. In all cases, the same random remapping is performed for a constant communication pattern. In Figure 7, graphs for the costs of building a schedule and redistribution are shown. With only two edges per node, it is better to use intervals during the first phase. During the second phase, the threshold is larger (3 edges) but still reasonably small.

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

[1] We also believe that this is critical to improve cache performance in current microprocessors.



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

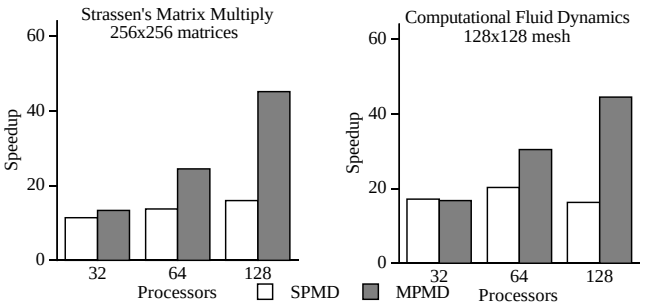
Figure 8: Example of Functional Parallelism

6 Functional and Data Parallelism

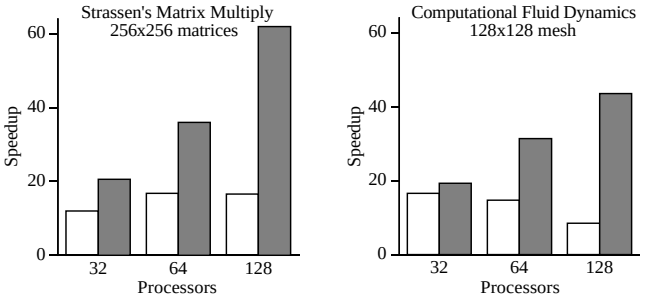
As mentioned in Section 2, a task graph, known as a Macro Dataflow Graph (MDG), is used to expose both the functional and data parallelism available in a program. The MDG for a given program is a weighted DAG with nodes representing routines in the program and edges representing precedence constraints among these routines. In the MDG, data parallelism information is implicit in the weight functions of the nodes while functional parallelism is captured by the precedence constraints among nodes.

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 routine and depends on the number of processors used to execute the routine. Scheduling may also make it necessary to redistribute an array between the processors allocated to a pair of routines. The time required for data redistribution depends on both the number of processors used by each routine as well as the final and initial distributions of the data in each routine.

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. In Figure 8a, a MDG with three nodes N_1, N_2, and N_3 is shown along with the processing costs and efficiencies of the nodes as a function of the number of processors. 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 8b. The first scheme exploits pure data parallelism, i.e., all routines use four processors. The second scheme exploits both functional and data parallelism, i.e., routines N_2 and N_3 execute concurrently and use two processors each. As



(a) CM-5



(b) Paragon

Figure 9: SPMD/MPMD Performance Comparison

shown by this example, good allocation and scheduling can improve program execution.

The allocation and scheduling algorithms in PARADIGM are based on the mathematical form 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 [20]. 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 shown to be theoretically bounded by a factor of the optimal value of the finish time. In practice, this factor is found to be small [20].

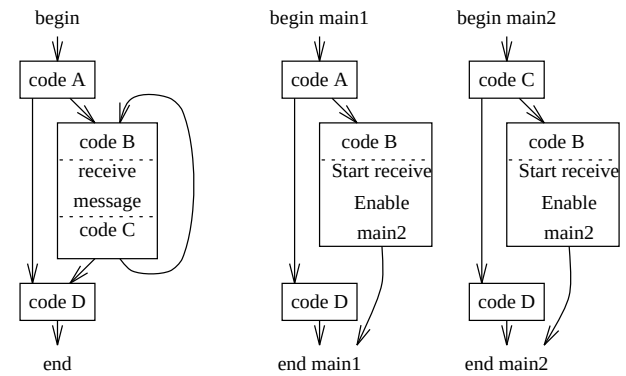
In Figure 9, the performance of the allocation and scheduling approach is compared to that of a pure

data-parallel approach. Speedups are measured on both the PARAGON and CM-5 for a pair of applications. Performance using the allocation and scheduling approach is identified as "MPMD" while 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 large processor systems, the performance of the programs is improved by a factor of about two to three times when comparing the MPMD execution to SPMD. These results demonstrate the utility of the allocation and scheduling approach.

7 Multithreading

Transforming message-passing code, such as that resulting from the previous compilation process, to message-driven code involves two main steps: the number of partitions are selected such that there are more total threads than processors, and receive operations are moved to the end of each thread. In the message-driven model, receive operations return control to the multithreading run-time system (MRTS) so that another waiting thread can be activated. This is simple for code without conditionals but becomes more complex when conditionals and loops are included. Although this paper only presents the transformation for converting *while* loops to message-driven code, similar transformations can be made on other control structures.

The transformation of the *while* loop is shown in Figure 10. Figure 10a shows the control flow graph of a message-passing program containing a receive in a *while* loop. Figure 10b shows the transformed code. In Figure 10b, *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.



(a) Original Control Flow Graph (b) Transformed Control Flow Graph

Figure 10: Transformation of the While Statement

After compilation into the SPMD programming model with blocking receives the transformation was performed on the following four scientific applications:

GS - Gauss-Seidel Iterative Solver
QR - 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 11 shows the speedup of the message-driven code with varying numbers of threads per processor compared to the single threaded SPMD code. The message-driven versions of the code outperform the SPMD versions in all cases except IMPL-1D. The applications that do show improvement show it for some number of threads greater than one, but the number of threads that produces the maximum speedup varies. For IMPL-1D, multithreading causes a significant increase in communications costs that cannot be overcome by multithreading.

Multithreading improves the runtime of certain applications. The message-driven model simplifies the implementation of the MRTS, and the basic algorithm for conversion is fairly straightforward [11].

8 Conclusions

PARADIGM is a multifaceted 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 compiler and run-time techniques. For programs which exhibit large grain functional parallelism, the compiler can also increase performance through proper resource allocation. 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.

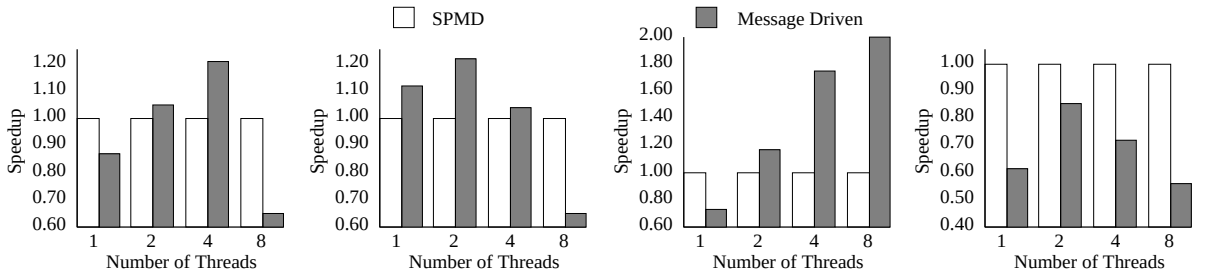


Figure 11: Speedup of Message Driven Threads on the CM-5

References

[1] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proc. of the ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation*, pages 126–138, Albuquerque, NM, June 1993.

[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. *Proc. of the 7th ACM Int'l Conf. on Supercomputing*, pages 351–360, July 1993.

[3] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. In *Proc. of the 3rd Work. on Compilers for Parallel Computers*, pages 145–164, 1992.

[4] I. Foster and K. M. Chandy. FORTRAN M: A Language for Modular Parallel Programming. Tech. Report MCS-P327-0882, Argonne Nat'l Lab., Argonne, IL, 1992.

[5] G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM 3.0 User's Guide and Reference Manual*. Oak Ridge Nat'l Lab., Oak Ridge, TN, Feb. 1993.

[6] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A Portable Instrumented Communication Library, C reference manual. Tech. Report ORNL/TM-11130, Oak Ridge Nat'l Lab., Oak Ridge, TN, July 1990.

[7] M. Gupta and P. Banerjee. PARADIGM: A Compiler for Automated Data Partitioning on Multicomputers. In *Proc. of the 7th ACM Int'l Conf. on Supercomputing*, Tokyo, Japan, July 1993.

[8] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0. Tech. Report CRPC-TR92225, Center for Research on Parallel Computation, Rice Univ., Houston, TX, May 1993.

[9] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[10] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *Proc. of the 6th ACM Int'l Conf. on Supercomputing*, Washington D.C., July 1992.

[11] J. G. Holm, A. Lain, and P. Banerjee. Compilation of Scientific Programs into Multithreaded and Message Driven Computation. In *Proc. of the 1994 Scalable High Performance Computing Conf.*, pages 518–525, Knoxville, TN, May 1994.

[12] A. Lain and P. Banerjee. Techniques to Overlap Computation and Communication in Irregular Iterative Applications. In *Proc. of the 8th ACM Int'l Conf. on Supercomputing*, pages 236–245, Manchester, England, July 1994.

[13] A. Lain and P. Banerjee. Exploiting Spatial Regularity in Irregular Iterative Applications. submitted to *Frontiers '95: The 5th Symp. on the Frontiers of Massively Parallel Computation*, 1995.

[14] J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays. In *Frontiers '90: The 3rd Symp. on the Frontiers of Massively Parallel Computation*, College Park, MD, Oct. 1990.

[15] Message-Passing Interface Forum. Document for a Standard Message-Passing Interface. Tech. Report CS-93-214, Univ. of Tennessee, Knoxville, TN, 1993.

[16] Office of Science and Technology Policy. *Grand Challenges 1993: High Performance Computing and Communications*. Nat'l Science Foundation, 1992.

[17] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee. Compiler Optimizations for Distributed Memory Multicomputers used in the PARADIGM Compiler. In *Proc. of the 23rd Int'l Conf. on Parallel Processing*, pages II:1–10, St. Charles, IL, Aug. 1994.

[18] 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 *Proc. of the 18th Int'l Conf. on Parallel Processing*, pages II:39–48, St. Charles, IL, Aug. 1989.

[19] R. Ponnusamy and J. Saltz. *A Manual for the CHAOS Runtime Library*. Univ. of Maryland, IACS, May 1994.

[20] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers. In *Proc. of the 23rd Int'l Conf. on Parallel Processing*, pages II:116–125, St. Charles, IL, Aug. 1994.

[21] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and Runtime Compilation. Tech. Report ICASE 90-59, Inst. for Computer Applications in Science and Eng., Hampton, VA, Sept. 1990.

[22] E. Su, D. J. Palermo, and P. Banerjee. Automating Parallelization of Regular Computations for Distributed Memory Multicomputers in the PARADIGM Compiler. In *Proc. of the 22nd Int'l Conf. on Parallel Processing*, pages II:30–38, St. Charles, IL, Aug. 1993.

[23] E. Su, D. J. Palermo, and P. Banerjee. Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers. In *Proc. of the 1994 Int'l Conf. on Parallel Archs. and Compilation Techniques*, pages 123–132, Montréal, Canada, Aug. 1994.

[24] J. Subhlok, J. M. Stichnoth, D. R. O'Halloran, and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. In *Proc. of the 4th ACM SIGPLAN Symp. on Principles & Practices of Parallel Prog.*, pages 13–22, San Diego, CA, May 1993.