Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-Memory Multicomputers

Ernesto Su, Antonio Lain, Shankar Ramaswamy,
Daniel J. Palermo, Eugene W. Hodges IV, and Prithviraj Banerjee

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 West Main St., Urbana, IL 61801, USA
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. A previous implementation of the compiler based on the PTD representation allowed symbolic array sizes, affine loop bounds and array subscripts, and variable number of processors, provided that arrays were single- or multi-dimensionally `block` distributed. The techniques presented here extend the compiler to also accept multi-dimensional `cyclic` and `block-cyclic` distributions within a uniform symbolic framework. These extensions demand more sophisticated symbolic manipulation capabilities. A novel aspect of our approach is to meet this demand by interfacing PARADIGM with a powerful off-the-shelf symbolic package, *Mathematica*™. This paper describes some of the *Mathematica*™ routines that performs various transformations, shows how they are invoked and used by the compiler to overcome the new challenges, and presents experimental results for code involving `cyclic` and `block-cyclic` arrays as evidence of the feasibility of the approach.

1 Introduction

Distributed-memory multicomputers 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 among 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 IBM fellowship and a Fulbright/MEC fellowship. We are also grateful to the Argonne National Lab for providing access to their machines.

The PARADIGM project at the University of Illinois addresses this problem by developing an automated means to parallelize and optimize sequential programs for efficient execution on multicomputers. It uses *Parafrase-2* [21] as a preprocessing platform to parse the sequential program into an intermediate representation, to analyze the code and generate flow, dependence, and call graphs, and to perform transformations such as constant propagation and induction variable substitution. Some of the other major research efforts in this area include Fortran D [16], Fortran 90D [10], the SUIF compiler [2], and the SUPERB compiler [11].

In addition to the traditional compiler optimizations to distribute computations and to reduce communication overheads, PARADIGM is unique in its ability to: (1) optionally parse High Performance Fortran (HPF) directives [17]; (2) perform automatic data distribution for regular computations, conventionally only specified through user directives; (3) generate high-level communication primitives; (4) optimize communication for regular computations; (5) support irregular computations using a combination of compile-time analysis and run-time support; (6) exploit functional and data parallelism simultaneously; and (7) generate multithreaded message-driven code to tolerate communication latencies. Current efforts in the project aim at integrating all of these features into the same framework [5]. Figure 1 shows a functional illustration of how we envision the complete PARADIGM compilation system. The compiler accepts either a sequential FORTRAN 77 or HPF program and produces a parallel program optimized for a target machine.

Previously, PARADIGM was implemented using Processor Tagged Descriptors [22] (PTD) as the underlying data structure to provide a uniform representation to describe both distributed arrays and partitioned loops. This implementation allowed symbolic array sizes, multi-dimensional distributions, variable number of processors, and affine loop bounds and array subscripts, but the distribution type was limited to `block`. Later extensions to PTD proved effective for purely `cyclic` distributions.

This paper presents techniques to support `block-cyclic` distributions, in addition to `block` and `cyclic`, all within a unified framework. The techniques are based on the Fourier-Motzkin elimination method [6] extended with many symbolic capabilities required by the compiler. A novel aspect of our approach is to meet the increasing demands for more sophisticated symbolic manipulations by exploiting the well-established functionality of existing powerful symbolic software, in our case, *Mathematica*. Just as most parsers today

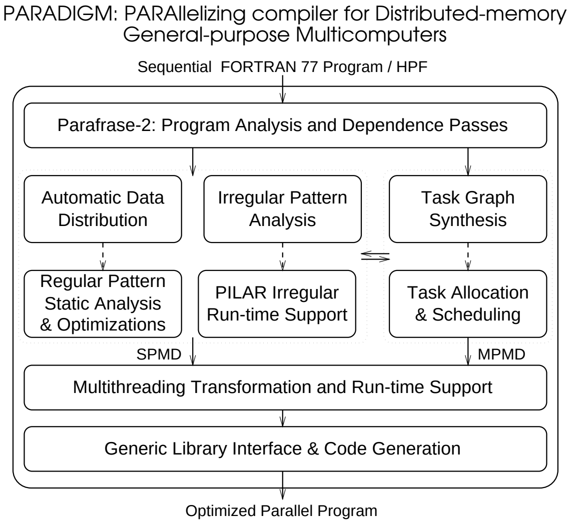

Figure 1: PARADIGM Compiler Overview

are implemented using stable and wide-spread tools like *yacc* and *lex* instead of being built from scratch, we argue that many parallelizing compilers in the future, in particular research prototypes developed in academic environments, will rely on off-the-shelf symbolic packages for most of their symbolic manipulations. Several reasons account for this:

- New compilers are required to perform increasingly complex symbolic tasks [9, 15, 22].
- As compiler algorithms become more complex, there is a need to focus more on their designs rather than on their implementation details.
- Many powerful symbolic packages are readily available (e.g., *Mathematica*, *Maple*), and some provide external interfaces (e.g., *MathLink*) with reasonable performance and robustness.

Clearly, the performance is not comparable with a C implementation optimized for special cases, but compilation time has not been a major problem in our experience. Furthermore, it is not a good practice to fully tune compiler features until their impact on optimizing real codes has been testes [8]. The approach presented in this paper allows quick prototyping and testing of new algorithms with real programs. Once the effectiveness of an algorithm is verified, then the critical features affecting efficiency and compilation time can be identified and rewritten in C.

The rest of this paper is organized as follows. Section 2 describes some of the *Mathematica* packages that we have written to meet the symbolic demands of the compiler. Section 3 shows how PARADIGM performs computation partitioning and communication generation by building inequality expressions from the source program and passing them to the Fourier-Motzkin package in *Mathematica* to obtain necessary symbolic expressions. Some results for two programs involving `cyclic` and `block-cyclic` array distributions, respectively, appear in Section 4. Finally, concluding remarks are made in Section 5.

2 Mathematica Tools in PARADIGM

This section describes a set of tools that we have built using *Mathematica* and shows how PARADIGM, which uses an internal representation based on Parafrase-2, is interfaced to *Mathematica* via *MathLink*. These tools are the result of only three man-months and represent roughly 1000 lines of *Mathematica* code and 700 lines of C code. The key to this quick turnaround is the extensive use of built-in *Mathematica* functionality as well as public domain *Mathematica* packages.

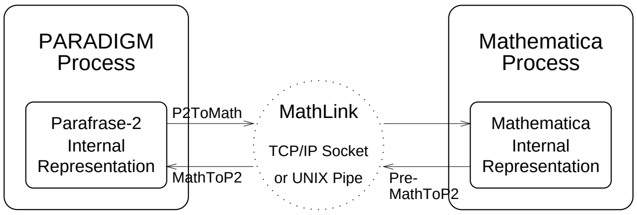

Figure 2: Linking PARADIGM with *Mathematica*

2.1 Linking PARADIGM with Mathematica

An overview of the interactions between PARADIGM and *Mathematica* is shown in Figure 2. This follows a typical client/server model in which the server (*Mathematica*) waits for a "symbolic request," processes it, and returns the result to the client. Both run as separate UNIX processes and communicate with each other using *MathLink*. The underlying communication mechanism can be either UNIX pipes, when they are running on the same machine, or TCP/IP sockets for remote execution.

Efficient forward and reverse conversion routines transfer data between PARADIGM's internal representation (based on Parafrase-2) and *Mathematica* expressions (*P2ToMath* and *MathToP2* in the figure). Conversion time is kept to a minimum using several techniques:

- The structure of the expression is never lost. We use *MathLink* API function calls to send and receive a tokenized stream.
- Variable names are transformed so that they encode a pointer to the symbol table. This eliminates symbol table look-ups when expressions return from *Mathematica*.
- *Mathematica* also pre-processes the result (*PreMathToP2* in the figure) so that functions have the correct number of arguments and easy-to-decode names following a preset convention.

An Example: Simplifying Loop Bounds To illustrate the interactions between PARADIGM and *Mathematica* we use a simple example, namely the simplification of loop bounds. After certain loop transformations, such as loop normalization, bound expressions can become very complex, and in particular, when arrays with multiple levels of indirection are involved, it is difficult for conventional compilers to "clean up the mess."

Figure 3 shows our solution to this problem using *Mathematica*. The PARADIGM process first opens a connection with the *Mathematica* process and then traverses the hierarchical representation of the loop nest and finds the lower and upper bound expressions. Each of these expressions is transformed into a *Mathematica* expression using *P2ToMath*, and a *Mathematica* built-in function, *Simplify*, is applied to them

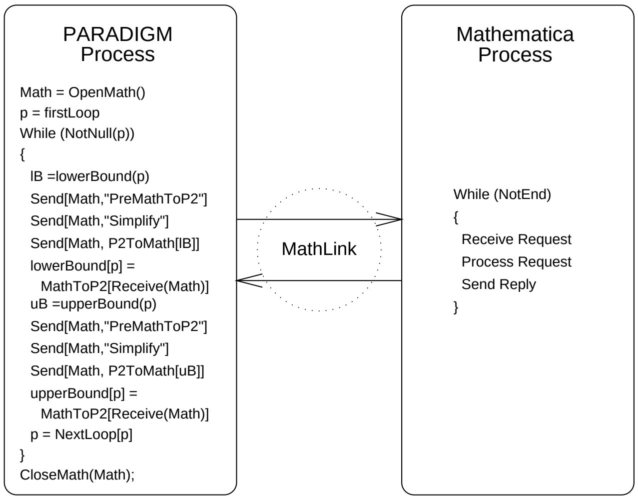

Figure 3: Simplifying Loop Bounds

by the server. The result is sent back to the client where it is transformed into a PARADIGM expression by *MathToP2* and inserted into the appropriate data structures.

A simple optimization that reduces the number of interactions between the processes is to generate a list with all the lower and upper bound expressions and send it for simplification as one stream. This is particularly important when the granularity of the *Mathematica* operations is small as it is in this case.

2.2 Symbolic Fourier-Motzkin Elimination (FME)

The Fourier-Motzkin method solves systems of linear inequalities with real variables using an elimination method. In the context of compilers it has important applications, including data dependence testing [19], inter-procedural analysis and, of particular importance to us, loop transformations [7] and automatic generation of communication for multicomputers [2, 3].

Our implementation quite closely follows the algorithm described by U. Banerjee [6] but also includes some of the conditions stated by M. Ancourt [4] to ensure that the integer projection is the same as the real projection. A fully symbolic implementation takes hardly 100 lines of *Mathematica* code, making extensive use of built-in functionality. Symbolic comparisons at every stage of the elimination process significantly reduce the number of redundant constraints and improve performance. This is particularly efficient when the compiler can provide information about the signs of symbolic constants, which is usually the case[1]. Other redundant constraints are eliminated using traditional methods [4].

To improve the quality of the output of our implementation of FME, we have extended the built-in simplification rules of *Mathematica*, providing extensive support for *floor*, *ceiling*, *min* and *max* operations. In particular, declaring unknown symbolic variables as integer and using the fact that integers are closed under certain operations $(+, -, \times)$, we can eliminate many unnecessary *floor* and *ceiling* operations. In addition, using a public domain package[2], expressions like $\frac{3a+4b+2}{3}$ can be determined as non-negative if a and b are first declared to be non-negative. This is used to eliminate operands in *min* and *max* operations by just subtracting operands and checking non-negativity of the result.

Since our compiler uses FME extensively to find various iteration sets (as will be described in Section 3), we shall briefly explain its interface. Its input consists of a set of inequalities $\mathcal{I}$ and a list of variables $\mathcal{V}$. The order of elimination of the variables in $\mathcal{V}$ is from the innermost to the outermost loop. Therefore, after elimination, a variable will only depend on enclosing loop variables. Thus, the solution set $\mathcal{R}$ that is returned consists of loop bounds for each variable in $\mathcal{V}$. These loop bounds can be used to scan the polytope defined by the original set of inequalities $\mathcal{I}$.

2.3 Loop Transformations

The functionality of a loop transformation pass is split between the PARADIGM process and the *Mathematica* process in the following way: the PARADIGM process identifies the loops involved in the transformation, checks that the transformation requested is valid using the data dependence and control flow graphs, establishes a connection with the *Mathematica* process, and sends it the relevant portions of the loop such as the loop bounds, array subscripts, and/or the entire loop body. The *Mathematica* process computes the new loop bounds after the transformation and a set of replacement rules to map the old loop variables into the new ones. It then returns the new loop bounds and loop body after applying the replacement rules. The PARADIGM process converts the results back into its internal representation.

Loop Normalization Loop normalization is typically performed as a pre-pass of the compiler analysis and reduces the lower bounds and strides to one. PARADIGM requests this transformation whenever the loop stride is not one. The implementation is trivial in *Mathematica* using built-in replacement and simplification rules. Note that affine subscripts and loop bounds remain affine after loop normalization.

Unimodular Loop Transformations Unimodular loop transformations provide a uniform view to many important transformations [6] (e.g. loop interchange, loop skewing, and loop reversal) and are a powerful tool to increase parallelism [7] and/or locality [23]. A unimodular loop transformation is described by a unimodular matrix, which is an integer matrix with a determinant of ± 1. In our *Mathematica* implementation, unimodular matrices are allowed to have symbolic terms. In particular, the unimodularity of a symbolic matrix is verified before its inverse is computed using built-in functions. Then, the corresponding loop transformations are applied to a set of loops by computing the new loop variables with the inverse unimodular matrix, generating a set of inequalities from the loop bounds and the new loop variables, and finally applying FME to obtain the new loop bounds. Roughly 20 lines of *Mathematica* code in addition to the Fourier-Motzkin package were required for performing unimodular loop transformations.

Redundant Access Elimination Transformations The method to be described in Section 3.2 to generate communication does not ensure that each element of an array is only transmitted once. Based on the early work of Gallivan et al [13], later refined by Ancourt [4], we can obtain a new set of loops that access each element at most once. The main idea behind this transformation is the decomposition

[1] Symbolic coefficients with unknown signs give rise to multiversion code.

[2] Thanks to Stephan Kaufmann at the Swiss Federal Institute of Technology for providing the NonNegativeQ package.

of the integer matrix that describes the access pattern into the product of other matrices with special characteristics. In our implementation, we used a *Mathematica* package[3] to obtain the Smith Normal Form and Hermite Column or Row Form of a matrix. Then, we can use the unimodular package described above to compute the new loop bounds. Currently PARADIGM is not taking advantage of this transformation, but it will in the near future.

2.4 Other Tools of Interest

In this section we briefly describe some other *Mathematica* tools that we have written that are useful in other areas of the compilation process.

Integer Area Estimation For applications like automatic data partitioning [14] or static buffer allocation it is crucial to have compile-time estimates of computation and/or communication requirements. Good estimates require the computation of the integer volume of a polyhedron, but this is in general too complex. Fortunately, in most practical cases we only need an estimate of this volume, and heuristics like the one proposed in [1] are sufficient. We have a preliminary 2-D implementation to estimate the integer area, which extends the work in [1] by handling affine loop bounds. The integer area is approximated by the "real" area plus half the points in the boundary. Using a *Mathematica* package[4], we obtain all the vertices of the polyhedron from the loop inequalities. Then, using built-in functionality, the vertices are ordered clock-wise and the "real" area is obtained through triangulation. Finally, the perimeter is computed by applying the GCD rule to consecutive points.

We can also apply this method in cases when symbolic terms only scale the polyhedron without changing its shape. In that case, we instantiate the symbolic terms for several values, compute the integer area with the previous method for each instantiation, and obtain an interpolating polynomial with the symbolic variable for these values using built-in functionality.

In cases where symbolic terms change the shape of the polyhedron, we can apply FME to obtain a bounding box as described in [4]. Assuming an n-dimensional space and using FME to project $n-1$ dimensions, we can obtain real bounds for the remaining dimension. Rotating the dimensions and repeating this process we can compute bounds for every dimension.

We are currently working on extensions to handle more than two dimensions and symbolic terms more effectively.

Graphic Visualization The built-in plotting functions in *Mathematica* are used to visualize iteration space and array access patterns in two or three dimensions. By merging plots for different processors (using different colors for each), the sets for a particular communication or the load distribution after partitioning a computation can be shown. Masks in plotting functions can be used to represent non-convex regions or filter certain types of points. By combining these plotting functions with FME, any convex set defined by linear inequalities can be visualized. This tool proved to be invaluable for debugging the compiler.

[3]Thanks to Brian L. Evans at Georgia Tech for providing the LatticeTheory package.

[4]Thanks to Komei Fukuda and Ichiro Mizukoshi of University of Tsukuba, Tokyo for providing the n-dimensional vertex enumeration package.

C++ Code Generation We have extended *Mathematica*'s capabilities to output structured C/C++ loops, conditionals, local declarations, and function headers and calls from *Mathematica* expressions. Together with the built-in function *Splice* and a template file, it is relatively easy to generate simple C++ code. We are using this capability to generate inspector code for irregular problems and link it with PARADIGM's irregular run-time support library, PILAR [18].

3 Compilation Techniques

This section will explain how PARADIGM compiles a loop nest containing computations on regularly distributed arrays. The arrays are allowed to have an arbitrary number of dimensions, each of which can have a `block`, `cyclic`, or `block-cyclic` distribution, or be replicated along a processor dimension, or be sequentialized on a single processor. The array subscripts and loop bounds are allowed to be affine functions of index variables of enclosing loops. Because of the symbolic nature of the approach, the number of processors need not be specified at compile time, allowing the resulting program to execute on an arbitrarily sized system. For simplicity, global references are not translated into local address spaces and, therefore, arrays are not scaled down as more processors are used. We are currently looking for an efficient way that can handle this local translation effectively and uniformly for all supported array distributions.

The data distribution and alignment of each array dimension is either automatically generated by PARADIGM [14] or provided by the user through standard HPF directives. Given the data distributions, the compiler still must carry out two major tasks, namely the partitioning of the computation across processors and the generation of communication code to transfer data among processors. How the compiler performs these tasks is the main focus of this section.

3.1 Computation Partitioning

The key to the symbolic analyses and transformations required for both of the aforementioned tasks is the use of the *Mathematica* tools described previously. The symbolic FME package is the main engine used to generate various iteration sets for different data access patterns.

To exploit parallelism in the source program, a compiler must somehow distribute computations across processors. PARADIGM partitions loops using the *owner computes* rule: a processor p only executes those iterations for which the left-hand side (*lhs*) array reference of an assignment statement is local to (owned by) p. Since determining ownership for each access at run time is prohibitively costly, the compiler, through a process called *loop bounds reduction* [16], derives new loop bounds confined to only p's local iterations. This is to say that the new loop must only traverse the ACCESS set [22] of the *lhs* reference. Consider the loop:

$$
\begin{array}{l}
\text{do } i = L, U \\
\quad \cdots A(2i+3) \cdots \\
\text{end do}
\end{array}
$$

where $A(l:u)$ is `block` distributed on P processors indexed by p, $0 \le p \le P-1$. By definition, $\textsc{Access}(A(s(i)), p)$ is the set of all iterations i such that $A(s(i))$ is stored in p. This set is obtained from the polytope $\mathcal{R}$ returned by FME when called with the inequalities $\mathcal{I}$ constraining the loop variable i, which is also the polytope's only axis variable in this example ($\mathcal{V} = i$). Two types of inequalities can be distinguished: *loop* inequalities and *data* inequalities. The former arises from loop bounds and specifies constraints on the loop variables and the relationships among them, while the latter is due to data distribution and describes the relationships between processor coordinates and subscript functions (hence loop variables, unless the subscript is a constant). In this example,

$$\mathcal{I} = \left\{ \begin{array}{ll} L \leq i \leq U & \text{(loop inequality)} \\ bp + l \leq 2i + 3 \leq & \\ \quad bp + l + b - 1 & \text{(data inequality)} \end{array} \right\}$$

where $b = \lceil \frac{u-l+1}{P} \rceil$ is the *block size* of distribution. FME returns:

$$\begin{aligned} \mathcal{R} &= \left\{ \left\lceil \max\left(L, \frac{-3+l+bp}{2}\right) \right\rceil \leq i \leq \right. \\ &\qquad \left. \left\lfloor \min\left(U, \frac{-4+b+l+bp}{2}\right) \right\rfloor \right\} \\ &= \textsc{Access}(A(s(i)), p) \end{aligned}$$

The symbolic capabilities of *Mathematica* allow FME to treat b, p, P, L, U, l, and u as literals, producing expressions that are parameterized by them, which have many advantages:

- The same code runs on every processor p, regardless of its location in the mesh.
- The number of processors (P) in the mesh need not be known at compile time.
- The values of the loop bounds, L and U, may be input at run time.
- The array bounds, l and u, need not be compile-time constants.

The same framework applies to `block-cyclic` distributions. An array with a `block-cyclic` distribution with block size b (written `cyclic`(b)) can be viewed as a 2-D $b \times N$ array on each processor p, where $N = \left(\left\lfloor \frac{u-l-bp}{bP} \right\rfloor + 1\right)$ is the number of blocks that p has. The blocks are enumerated with a *block number* n, $0 \leq n \leq N-1$, and the heads of any two consecutive blocks of the same processor are at a distance bP apart. The nth block in p therefore contains array elements from $bp + l + bnP$ to $bp + l + bnP + b - 1$, inclusive[5]. For the same loop as before, the input to FME becomes:

$$\begin{aligned} \mathcal{I} &= \left\{ \begin{array}{ll} L \leq i \leq U & \text{(loop ineq.)} \\ 0 \leq n \leq \left\lfloor \frac{u-l-bp}{bP} \right\rfloor & \text{(data ineq.)} \\ bp + l + bnP \leq 2i + 3 \leq & \\ \quad bp + l + bnP + b - 1 & \text{(data ineq.)} \end{array} \right\} \\ \mathcal{V} &= n, i \end{aligned}$$

and the output is:

$$\mathcal{R} = \left\{ \begin{array}{l} \left\lceil \max\left(0, \frac{4-b-l+2L-bp}{bP}\right) \right\rceil \leq n \leq \left\lfloor \frac{3-l-bp+2U}{bP} \right\rfloor \\ \left\lceil \max\left(L, \frac{-3+l+bp+bnP}{2}\right) \right\rceil \leq i \leq \\ \qquad \left\lfloor \min\left(U, \frac{-4+b+l+bp+bnP}{2}\right) \right\rfloor \end{array} \right\}$$

[5] Although one of the processors may have an incomplete last block, this causes no problems as the loop inequalities will provide a tighter upper bound.

Once $\mathcal{R}$ is found, Access(A(s(i)),p) is simply the set of i's in $\mathcal{R}$. A purely `cyclic` distribution is just `cyclic`(1), so this case is covered by setting $b = 1$ in the above expressions. These results easily extend to multi-dimensional arrays distributed on multi-dimensional processor meshes.

Some special cases are worth mentioning. If an array dimension $A(\ldots, l : u, \ldots)$ is replicated, then its data inequality is $l \leq s(i) \leq u$ and involves no processor indices. If an entire array dimension is sequentialized (or collapsed) on a particular processor r, then instead of a data inequality, the mask "IF $p = r$ THEN" is applied to the Access set.

To perform loop bounds reduction, PARADIGM just needs to extract the loop inequalities from the the original loop bounds and the data inequalities corresponding to the *lhs* reference and its distribution, and send them to FME. Then, it uses the polytope returned to construct the new loop.

Using the previous results for the Access set of a `block` distributed array A, the sequential loop:

do $i = L, U$
  $A(2i+3) = \cdots$
end do

becomes:

$p = \text{myp}()$
$b = \lceil \frac{u-l+1}{P} \rceil$
do $i = \left\lceil \max\left(L, \frac{-3+l+bp}{2}\right) \right\rceil, \left\lfloor \min\left(U, \frac{-4+b+l+bp}{2}\right) \right\rfloor$
  $A(2i+3) = \cdots$
end do

after loop bounds reduction, where myp() returns the processor's index in the mesh. Similarly, for a `cyclic`(b) distribution the new loop is:

$p = \text{myp}()$
do $n = \left\lceil \max\left(0, \frac{4-b-l+2L-bp}{bP}\right) \right\rceil, \left\lfloor \frac{3-l-bp+2U}{bP} \right\rfloor$
  do $i = \left\lceil \max\left(L, \frac{-3+l+bp+bnP}{2}\right) \right\rceil,$
      $\left\lfloor \min\left(U, \frac{-4+b+l+bp+bnP}{2}\right) \right\rfloor$
    $A(2i+3) = \cdots$
  end do
end do

When there are multiple *lhs* array references, a conservative union $\mathcal{U}$ of their Access sets is used to form the reduced loop bounds, while each assignment statement is masked with the corresponding Access set of its *lhs*. For example, in the following loop:

do $i = L, U$
  $A(2i+3) = \cdots$
  $B(i-1) = \cdots$
end do

let both A and B have a `block-cyclic` distribution, and let their Access sets be $\{L_{n_A} \leq n \leq U_{n_A}, L_{i_A} \leq i \leq U_{i_A}\}$ and $\{L_{n_B} \leq n \leq U_{n_B}, L_{i_B} \leq i \leq U_{i_B}\}$, respectively. Their union is:

$$\mathcal{U} = \left\{ \begin{array}{l} \min(L_{n_A}, L_{n_B}) \leq n \leq \max(U_{n_A}, U_{n_B}) \\ \min(L_{i_A}, L_{i_B}) \leq i \leq \max(U_{i_A}, U_{i_B}) \end{array} \right\}$$

and so the loop becomes:

do $n = \min(L_{n_A}, L_{n_B}), \max(U_{n_A}, U_{n_B})$
  do $i = \min(L_{i_A}, L_{i_B}), \max(U_{i_A}, U_{i_B})$
    if $((L_{n_A} \le n \le U_{n_A})$ .and. $(L_{i_A} \le i \le U_{i_A}))$ then
      $A(2i+3) = \cdots$
    end if
    if $((L_{n_B} \le n \le U_{n_B})$ .and. $(L_{i_B} \le i \le U_{i_B}))$ then
      $B(i-1) = \cdots$
    end if
  end do
end do

Since `block` and `cyclic` distributions are special cases of `block-cyclic`, this method works for all types of regular distributions. In particular, if some of the *lhs* terms are `block` distributed, then before finding their ACCESS sets, they are first cast into a `block-cyclic` form by adding a bnP term to the bounds in their data inequality, and adding an extra data inequality, $0 \le n \le 0$. However, if all of the *lhs* terms are `block` distributed, then this transformation is not necessary.

3.2 Communication Generation

If a processor p does not own all of the elements of a right-hand side (*rhs*) array reference R required by a statement that it executes, then this data must be sent from its owner, say q, to p via inter-processor communication. To reduce communication overheads, array elements can be combined into a single larger message instead of being communicated individually. This optimization is generally known as *message vectorization* [16].

The approach is the basically similar to the work of Ancourt [4] which describes regions requiring communication using a set of linear inequalities and generates loops to scan these regions through a Fourier-Motzkin projection. Dependence analysis done by Parafrase-2 is used to determine the communication point, which is the outermost loop at which the combining can be applied.

Therefore, the main role of the compiler is to obtain relevant linear inequalities from the loop bounds, data decompositions, and array references in the source program, send these inequalities to FME, and from its results generate the scanning loops to pack/send and receive/unpack, and insert this code at the communication point.

The set of iterations for which a processor p must receive elements of a *rhs* reference R from its owner processor q is called the COMM set of R, denoted COMM(R, p, q) (In previous work [22], this was referred to separately as the SEND set of q and the RECEIVE set of p). To obtain the COMM set, the inequalities $\mathcal{I}$ needed are the loop inequalities[6], the data inequalities (parameterized by p) of the *lhs* reference, and the data inequalities (parameterized by q) of the *rhs* reference R. The resulting polytope $\mathcal{R}$ is used to both pack/send data (by setting $q =$myp()) and receive/unpack data (by setting $p =$myp()).

This process is demonstrated using the HPF program in Figure 4. The processor grid is a $P_1 \times P_2$ mesh[7] (hence $0 \le p_1 \le P_1 - 1$ and $0 \le p_2 \le P_2 - 1$). The *lhs* array A is distributed by `cyclic(5)` on the first mesh dimension and sequentialized ($p_2 = 0$) on the second mesh dimension; i.e., only the processors whose second coordinate p_2 is 0 will own parts of A. The first dimension of the *rhs* array B is distributed by `cyclic(3)` on the first mesh dimension, while the second array dimension is `cyclic(7)` on the second mesh dimension.

[6] From loops nested below the communication point only; index variables of outer loops are treated as symbolic constants.

[7] The usage of the processors declaration in the program is not standard, but it simplifies the specification of a variable number of processors in both mesh dimensions.

```
      real A(170)
      real B(120,120)
!HPF$ processors P(P1,P2)
!HPF$ template T(170,2)
!HPF$ align A(k) with T(k,1)
!HPF$ distribute T(cyclic(5), block)      onto P
!HPF$ distribute B(cyclic(3), cyclic(7)) onto P
      do i = 3,40
        do j = 2,i-1
          A(4i+5) = B(2i+j-1, 3i-2j+1)
        end do
      end do
```

Figure 4: Example Loop

First, the compiler uses data dependence analysis to determine that communication can take place outside the entire loop nest. Then, it calls FME with inequalities derived from the loop (as explained below), and extracts the COMM set from the solution $\mathcal{R}$ returned to construct the communication code to pack/send and receive/unpack data. The input sent to FME is:

$$\mathcal{I} = \left\{ \begin{array}{ll} 3 \le i \le 40 & (i\text{-loop}) \\ 2 \le j \le i-1 & (j\text{-loop}) \\ 0 \le n \le \lfloor \frac{170-1-5p_1}{5*4} \rfloor & (lhs) \\ 5p_1 + 1 + 5*4n \le 4i+5 \le & \\ \quad 5p_1 + 1 + 5*4n + 5 - 1 & (lhs) \\ 0 \le m_1 \le \lfloor \frac{120-1-3q_1}{3*4} \rfloor & (rhs_{d1}) \\ 3q_1 + 1 + 3*4m_1 \le 2i+j-1 \le & \\ \quad 3q_1 + 1 + 3*4m_1 + 3 - 1 & (rhs_{d1}) \\ 0 \le m_2 \le \lfloor \frac{120-1-7q_2}{7*2} \rfloor & (rhs_{d2}) \\ 7q_2 + 1 + 7*2m_2 \le 3i - 2j + 1 \le & \\ \quad 7q_2 + 1 + 7*2m_2 + 7 - 1 & (rhs_{d2}) \end{array} \right\}$$

$$\mathcal{V} = n, m_1, m_2, i, j$$

The loop inequalities come directly from the loop bounds. A pair of data inequalities comes from each of the three array dimensions involved (one from *lhs* and two from *rhs*); each pair consists of an inequality bounding the block number and one bounding the subscript function. The block numbers for the first and second dimensions of the *rhs* are m_1 and m_2, respectively. Since the *rhs* determines the sender (q_1, q_2), the processor coordinates involved in these inequalities are q_1 and q_2. Similarly, the block number for the *lhs* is n, and the processor coordinate involved is only p_1 because the *lhs* determines the receiver (p_1, p_2) and A is distributed only along the first processor dimension. Although p_2 is not involved in the inequalities, the compiler takes into account the fact that only processors with coordinate $p_2 = 0$ own parts of the *lhs* and hence are potential receivers, and generates code accordingly (shown in Figure 5). The loops

```
{PACK/SEND phase: processor (q1, q2)
 sends to processor (p1, p2 = 0)}
   q1 = myp1()
   q2 = myp2()
   p2 = 0
   do 20 p1 = 0, P1 - 1
    if ( (p1 .ne. q1) .or. (p2 .ne. q2) ) then
     len = 0
     do 10 n = ceiling(max(0, real(4 - 5p1 + 4q1)/(5P1),
        real(16 - 35p1 + 24q1 + 28q2)/(35P1),
        real(48 - 35p1 + 28q2)/(35P1),
        real(16 - 15p1 + 28q2)/(15P1),
        real(12 - 5p1)/(5P1))),
       floor(min(real(3356 - 15p1 - 112q2)/(15P1),
        real(164 - 5p1)/(5P1), real(236 - 5p1 - 4q2)/(5P1),
        real(1124 - 5p1 - 24q1)/(5P1),
        real(284 - 5p1 - 7q2)/(5P1),
        real(844 - 5p1 - 21q2)/(5P1)))
      do 10 m1 = ceiling(max(0, real(4 - 3q1)/(3P1),
         real(44 - 15p1 - 15nP1 - 24q1)/(24P1),
         real(-548 + 35p1 + 35nP1 - 24q1)/(24P1),
         real(-16 + 5p1 + 5nP1 - 18q1)/(18P1),
         real(-8 + 5p1 + 5nP1 - 6q1)/(6P1),
         real(2 - 9q1 + 14q2)/(9P1))),
        floor(min(real(39 - q1)/P1,
         real(1124 - 5p1 - 5nP1 - 24q1)/(24P1),
         real(-4 + 5p1 + 5nP1 - 4q1)/(4P1),
         real(-16 + 35p1 + 35nP1 - 24q1 - 28q2)/(24P1),
         real(276 - 6q1 - 7q2)/(6P1)))
       do 10 m2 = ceiling(max(0,
          real(-20 + 5p1 + 5nP1 - 28q2)/(28P2),
          real(-84 + 35p1 - 24m1P1 + 35nP1
            -24q1 - 28q2)/(28P2),
          real(7 - 6m1P1 - 6q1 - 7q2)/(7P2),
          real(-3 + m1P1 + q1 - 7q2)/(7P2))),
         floor(min(real(116 - 7q2)/(7P2),
          real(-16 + 15p1 + 15nP1 - 28q2)/(28P2),
          real(-16 + 35p1 - 24m1P1 + 35nP1
            -24q1 - 28q2)/(28P2),
          real(-2 + 9m1P1 + 9q1 - 14q2)/(14P2),
          real(276 - 6m1P1 - 6q1 - 7q2)/(7P2)))
        do 10 i = ceiling(max(3, real(4 + 7m2P2 + 7q2)/3,
           real(-4 + 5nP1 + 5p1)/4, 1 + m1P1 + q1,
           m2P2 + q2 + real(4 + 6m1P1 + 6q1)/7)),
          floor(min(40, real(5nP1 + 5p1)/4,
           real(2 + 3m1P1 + 3q1)/2, 4 + 7m2P2 + 7q2
           2 + m2P2 + q2 + real(6m1P1 + 6q1)/7))
         do 10 j = ceiling(max(2, 2 - 2i + 3m1P1 + 3q1,
            real(-6 + 3i - 7m2P2 - 7q2)/2)),
           floor(min(-1 + i, 4 - 2i + 3m1P1 + 3q1,
            real(3i - 7m2P2 - 7q2)/2))
          buffer(len) = B(2i + j - 1, 3i - 2j + 1)
          len = len+1
10   continue
     send(msgid(q1, q2), buffer, len*4, (p1, p2) )
    end if
20 continue
```

```
{RECEIVE/UNPACK phase: processor (p1, p2 = 0)
 receives from processor (q1, q2)}
   p1 = myp1()
   p2 = myp2()
   if (p2 .eq. 0) then
    do 40 q1 = 0, P1 - 1
     do 40 q2 = 0, P2 - 1
      if ( (p1 .ne. q1) .or. (p2 .ne. q2) ) then
       len = 0
       recv(msgid(q1, q2), buffer, BUFFERSIZE)
       do 30 n = L_n(P1, P2, p1, q1, q2),
          U_n(P1, P2, p1, q1, q2)
        do 30 m1 = L_m1(P1, P2, p1, q1, q2, n),
           U_m1(P1, P2, p1, q1, q2, n)
         do 30 m2 = L_m2(P1, P2, p1, q1, q2, n, m1),
            U_m2(P1, P2, p1, q1, q2, n, m1)
          do 30 i = L_i(P1, P2, p1, q1, q2, n, m1, m2),
             U_i(P1, P2, p1, q1, q2, n, m1, m2)
           do 30 j = L_j(P1, P2, p1, q1, q2, n, m1, m2, i),
              U_j(P1, P2, p1, q1, q2, n, m1, m2, i)
            B(2i + j - 1, 3i - 2j + 1) = buffer(len)
            len = len+1
30     continue
      end if
40  continue
   end if
```

```
{EXECUTION phase}
   if (p2 .eq. 0) then
    do 50 n = ceiling(real(12 - 5p1)/(5P1)),
      floor(real(164 - 5p1)/(5P1))
     do 50 i = ceiling(max(3,
        real(-4 + 5nP1 + 5p1)/4)),
       floor(min(40, real(5nP1 + 5p1)/4))
      do 50 j = 2, -1 + i
       A(4i + 5) = B(2i + j - 1, 3i - 2j + 1)
50  continue
   end if
```

Figure 5: SPMD Code for the Example Loop

in the receive/unpack phase have the same bound expressions as those in the pack/send phase, since both come from the same polytope. Therefore, we have only included their abbreviated bound expressions along with the variables of which these expressions are functions. The figure also includes the execution phase, which is the result of the loop bounds reduction procedure described previously.

In this example, the communication code essentially traverses the entire processor space since the program has an all-to-many communication pattern. However, the communication pattern is often not as complicated. For example, many stencil computations exhibit one-to-one (shift) communication patterns when their arrays are properly aligned. In such cases, it is useful to find receiving and sending *processor sets*. Then, the send phase only has to scan through the receiving processor set instead of the entire processor space. Similarly, the receive phase only scans through the sending processor set for candidate senders. The sending processor set can be found simultaneously with the COMM set in a single invocation to FME. To do this, FME is called with the same input that we used to find the COMM set as before, but with two additions. A *processor* inequality $0 \leq q \leq P-1$ is added to $\mathcal{I}$, and the variable q is *prepended* to the list $\mathcal{V}$, as q is now the first polytope axis. Note that this set is parameterized by the receiver p. The same steps can be followed to find the receiving processor set; simply replace q by p and proceed as before. Similarly, this set is parameterized by the sender q.

4 Results

Two small scientific programs, LU and POTENG, were compiled using the techniques described in the previous sections. The communication generation is not fully integrated with the rest of the compiler yet, so we manually added the communication code generated by *Mathematica*. LU is a standard LU matrix factorization code, and POTENG computes the potential energy in the molecular dynamic simulation program (MDG) of the Perfect Benchmarks [20]. These programs were chosen because cyclic or block-cyclic distributions are required to obtain reasonable performance. Both programs were run on an IBM SP-2 using MPIF [12].

LU has three 1024×1024 2-D arrays, two distributed in a column-cyclic manner (U and the input matrix) while the other one (L) in a row-cyclic manner. The computation consists mainly of two doubly-nested loops (one for L and the other for U), both enclosed in an outermost loop. In each iteration of the outermost loop the program computes a column of L and a row of U, where L and U are triangular matrices. Both operations are performed in parallel and with reasonable load balance due to the cyclic distributions. Communication occurs outside the doubly-nested loops but inside the outermost loop. The communication pattern involves a broadcast of a row and a column in each iteration of the outermost loop. The broadcast initiator, the size of the broadcast message, and the number of receiving processes depend on the outermost iteration. This makes it difficult to identify the broadcast. The speedup for up to 32 processors is shown in Figure 6. We can see in this figure that it is critical to identify the communication pattern as a broadcast, and therefore be able to use a MPI collective communication primitive instead of point-to-point communication. We are currently extending the high-level communication detection features of PARADIGM to handle such cases.

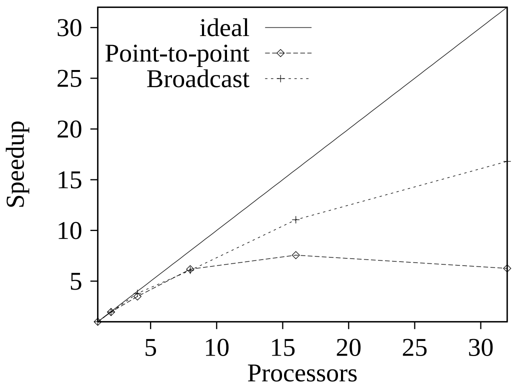

Figure 6: Speedup of LU

POTENG has six 1-D arrays, three of size N, where N is the number of molecules to be simulated, and the other three of size $3N$, where 3 is the number of atoms in a water molecule. A `cyclic` distribution is selected for the first three arrays while a `cyclic`(3) distribution (i.e., `block-cyclic` with block size of 3) for the second.

This subroutine has two main loops. The first one has statements of the form:

double precision $x(3N)$, $xm(N)$, $c1$, $c2$

do $i = 1, N$
  $xm(i) = c1 * x(3i-1) + c2 * (x(3i-2) + x(3i))$
  $\cdots$ {similar computations with ym and zm} $\cdots$

while the second one is a computationally intensive triangular loop. The `cyclic`(3) distribution becomes a natural choice to satisfy both communication and load balancing requirements (in agreement with the decision made by PARADIGM's automatic data partitioner [14]).

Prerequisite steps for the parallelization of the triangular loop are the following:

- Inlining three small subroutine calls.
- Fairly complex induction variable elimination.
- Array privatization of five small arrays.
- Detecting two simple reductions.

In our case we used the induction variable elimination of Parafrase-2 and manually performed the other steps. All of these steps are automatically performed by the Polaris compiler[8].

Communication operations generated by PARADIGM were combined outside the triangular loop, and reductions were performed using MPI collective communication primitives. Figure 7 plots the speedup for $N = 3000$ for up to 32 processors. The encouraging speedup shown in the figure is a good indication that the distributions chosen are correct and that our compilation techniques are effective.

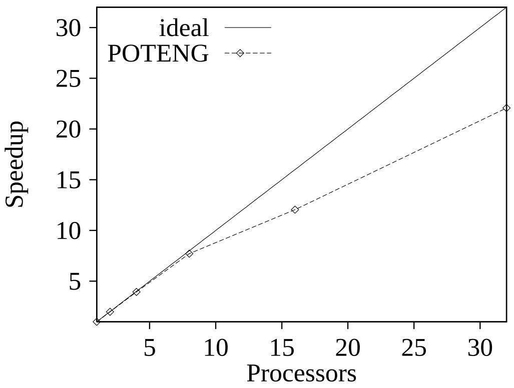

Figure 7: Speedup of POTENG

5 Conclusions and Future Work

The techniques presented in this paper allows PARADIGM to deal with all of the common regular data decompositions (`block`, `cyclic`, and `block-cyclic`) within a unified framework. By leveraging off the symbolic power of existing packages like *Mathematica*, PARADIGM has immediate access to symbolic capabilities that ultimately allows a much wider range of compilable programs than was possible before.

The work presented does not translate global references to the local address space of a processor; we are working on better ways to handle local translations uniformly for any of the supported data distributions. We are also extending previous work [14] on automatic detection of high-level communication to handle more complicated cases such as that encountered in LU.

References

[1] Agarwal, A., Kranz, D., and Natarajan, V. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Processing* (St. Charles, IL, Aug. 1993), pp. I:2–11.

[2] Amarasinghe, S. P., and Lam, M. S. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation* (Albuquerque, NM, June 1993), pp. 126–138.

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

[4] Ancourt, M. *Generation Automatique de Codes de Transfert pour Multiprocesseurs a Memoires Locales.* PhD thesis, Universite Paris VI, Mar. 1991.

[5] Banerjee, P., Chandy, J. A., Gupta, M., Holm, J. G., Lain, A., Palermo, D. J., Ramaswamy, S., and Su, E. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *Proceedings of the First International Workshop on Parallel Processing* (Bangalore, India, Dec. 1994), pp. 322–330.

[6] Banerjee, U. *Loop Transformations for restructuring compilers: The Foundations.* Kluwer Academic Publishers, Boston, MA, 1993.

[7] Banerjee, U. *Loop Transformations for restructuring compilers: Loop parallelization*, vol. II. Kluwer Academic Publishers, Boston, MA, 1994.

[8] Blume, W., and Eigenmann, R. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems 3*, 6 (Nov. 1992), 643–656.

[9] Blume, W., and Eigenmann, R. The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions. In *Proceedings of Supercomputing '94* (Washington D.C., 1994), pp. 528–537.

[10] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., and Ranka, S. 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, July 1993), pp. 351–360.

[11] Chapman, B., Mehrotra, P., and Zima, H. Programming in Vienna Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers* (1992), pp. 145–164.

[12] Franke, H., Hochschild, P., Pattnaik, P., and Snir, M. MPI-F: An Efficient Implementation of MPI on IBM-SP1. In *Proceedings of the 23rd International Conference on Parallel Processing* (St. Charles, IL, Aug. 1994), pp. III:197–201.

[13] Gallivan, K., Jalby, W., and Gannon, D. On the Problem of Optimizing Data Transfers for Complex Memory Systems. In *Proceedings of the Second ACM International Conference on Supercomputing* (Saint Malo, France, 1988).

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

[15] Haghighat, M. R. *Symbolic Analysis for Parallelizing Compilers.* PhD thesis, University of Illinois, Urbana-Champaign, IL, Sept. 1994.

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

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

[18] Lain, A., and Banerjee, P. Exploiting Spatial Regularity in Irregular Iterative Applications. In *Proceedings of the 9th International Parallel Processing Symposium* (Santa Barbara, CA, Apr. 1995). To appear.

[19] Maydan, D. E., Hennessy, J. L., and Lam, M. S. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada, June 1991), pp. 1–14.

[20] Perfect Club. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputing Applications 3*, 3 (Fall 1989), 5–40.

[21] Polychronopoulos, C. D., Girkar, M., Haghighat, M. R., Lee, C. L., Leung, B., and Schouten, D. 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, Aug. 1989), pp. II:39–48.

[22] Su, E., Palermo, D. J., and Banerjee, P. 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, Aug. 1994), pp. 123–132.

[23] Wolf, M. E. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, CA, Aug. 1992. CSL-TR-92-538.