AUTOMATING PARALLELIZATION OF REGULAR COMPUTATIONS FOR DISTRIBUTED-MEMORY MULTICOMPUTERS IN THE PARADIGM COMPILER*

Ernesto Su, Daniel J. Palermo, and Prithviraj Banerjee
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.
{ernesto, palermo, banerjee}@crhc.uiuc.edu

Abstract – Distributed-memory multicomputers such as the Intel iPSC/860, the NCUBE/2, the Intel Paragon and the Connection Machine CM-5 offer significant advantages over shared-memory multiprocessors in terms of cost and scalability. Unfortunately, to extract all the computational power from these machines, users have to parallelize their existing serial programs, which can be an extremely laborious process. One major reason for this difficulty is the absence of a single global shared address space. As a result, the programmer has to distribute code and data on processors and manage communication among tasks explicitly. Clearly there is a need for efficient parallelizing compiler support on these machines. The PARADIGM project at the University of Illinois addresses these problems by developing a fully automated means to translate serial programs for efficient execution on distributed-memory multicomputers. In this paper we discuss parallelization of regular computations using symbolic sets as a representation for data accesses and communication.

1. INTRODUCTION

One of the major advantages of distributed-memory multicomputers over shared-memory multiprocessors is that the former are much more easily scalable than the latter. Therefore, it is much easier to build massively parallel systems with distributed memory than with shared memory. However, lacking a global address space, distributed-memory machines are considerably more difficult to program than shared-memory machines. They rely on the programmer to perform important tasks whose quality directly affects the performance of the resulting program: to distribute data on different processors, and to implement interprocessor communication whenever a process needs data from another processor.

In recent years, significant research effort has been aimed at source-to-source parallel compilers for multicomputers that relieve the programmer from the task of communication generation, while the task of data partitioning remains a responsibility of the programmer [1, 2, 3, 4]. Based on the user-specified partitioning of data, these compilers can generate the corresponding parallel program for a multicomputer.

*This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096, and in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613.

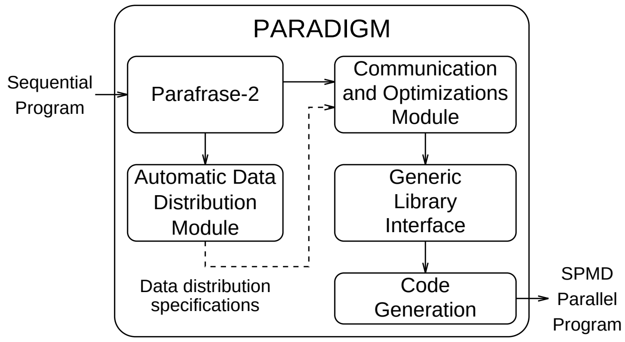


Figure 1: Compiler Overview

Our research aims at a compiler that can parallelize existing programs written in a common sequential language (currently FORTRAN 77) with little or no further input from the programmer. This means that the compiler must perform not only automatic communication generation, but also automatic data partitioning. Figure 1 shows a simplified view of the compiler, called PARADIGM (**PARA**llelizing compiler for **DI**stributed-memory **G**eneral-purpose **M**ulticomputers). Within the framework, *Parafrase-2* [5] is used as a preprocessing platform to parse the input program into an intermediate representation and to perform useful analysis and transformations. The Automatic Data Distribution Module determines how data is to be partitioned across processors and, based on this information, the Communication and Optimizations Module partitions the computations and generates communication accordingly. The compiler can also be easily retargeted to different machines through the use of a Generic Library Interface before code generation. The final output of PARADIGM is an SPMD (Single Program Multiple Data) parallel program with message passing.

Recently, the Automatic Data Distribution Module has been successfully developed and implemented by Gupta and Banerjee [6, 7]. It uses a constraint-based approach to choose data distributions that minimize both the computation and communication costs of a program for a particular target machine. For each target machine, there is a set of parameters describing relative computation and communication costs, so the decision algorithm is effectively isolated from the target architecture. This paper addresses the computation partitioning and communication generation aspects of PARADIGM.

Much research is being done to increase the functionality of PARADIGM. When fully implemented, it will be capable of performing all of the following tasks automatically:

- Generation of data partitioning specifications [6, 7].
- Partitioning of computations and generation of communication for data-parallel programs [8].
- Generation of high-level communication [9].
- Exploitation of functional parallelism [10, 11, 12].
- Support for multithreaded execution [13].

The rest of this paper is organized as follows. Section 2 describes the techniques used to analyze the access patterns of the input program. Section 3 outlines the approach taken in generation of communication using the information from the analysis phase. Section 4 presents a study of the performance of the compiler on FORTRAN 77 programs with regular computations to determine the applicability of our approach. Finally, conclusions are presented in Section 5.

2. ANALYSIS

Basically, our approach applies the *owner computes rule* which states that the processor owning a data item must perform all computations for that item. Any nonlocal data needed in the computation is obtained via interprocessor communication. A direct application of this rule without further optimizations leads to code with *runtime resolution* which explicitly computes the ownership and communication for each reference, resulting in extremely high computation and communication overheads. Using compile-time analysis and optimizations, it is possible to eliminate most of the overheads, generating much more efficient parallel programs [14].

2.1 Symbolic Sets

The computation partitioning, communication analysis, and optimization phases (all described later) require an efficient way of describing sets of iterations and regions of data. Our approach utilizes *symbolic sets*, which have many desirable characteristics.

The symbolic sets are represented as a lower and upper bound where each bound can be a single linear expression or a combination of two linear expressions (*min* or *max* operation, to represent masking operations on the sets). A multidimensional set has the following representation for each dimension:

$$\mathbf{set}_{dim} = [\mathit{lower\ bound} : \mathit{upper\ bound}[: \mathit{step}]\ [, \mathit{cycle}]]$$

$$\mathbf{bound} = \begin{cases} c_1 p + c_0 \\ min(c_1 p + c_0, d_1 p + d_0) \\ max(c_1 p + c_0, d_1 p + d_0) \end{cases}$$

where c_i, d_i, *step*, and *cycle* are all constants.

A single set representation is parameterized by the processor location (p) for each dimension of a virtual mesh, providing a uniform representation for every processor in the mesh, whether it is a boundary or an interior node. As a result, operations on the sets are very efficient because all processors in one or more dimensions can now be captured into a single symbolic set operation. In addition, symbolic sets and set operations are easily extended to an arbitrary number of dimensions, necessary for describing iteration sets in multiply nested loops as well as sections of multidimensional arrays.

Masks (*min*, *max*) in the set representation are used to express the upper and lower bounds of a given access. The optional *step* of a symbolic set corresponds to the addressing of elements within the range of the bounds. If not specified, then *step* is assumed to be one.[a] The *cycle* of a set is used to describe the replication of a block description through a cyclic distribution. Note that in global coordinates, the *cycle* can be viewed as a *mod* function on the iterations. In local coordinates, it can be viewed as an offset between replicated sub-block descriptors. In either case, the bound expressions for the transformations (to be defined later) are unaffected by the *cycle* parameter. For completeness, the *cycle* for a blocked distribution is the size of the entire space for the specific array dimension.

Both the data partitioning and the subscript reference affect the mapping of global iterations to the mesh of processors. Given a subscript reference $A(s(i))$ (with $s(i) = a_1 i + a_0$) as well as the block size b of the (cyclic or blocked) distribution, a symbolic set representing the *global iterations*[b] for which $A(s(i))$ is stored in a processor p can be constructed as:

$$\begin{aligned} a_1 i + a_0 &\in [bp + 1 : bp + b] \\ i &\in [\lceil lbound(p) \rceil : \lfloor ubound(p) \rfloor] \\ \text{where} \quad lbound(p) &= \left(\frac{b}{a_1}\right) p + \left(\frac{1 - a_0}{a_1}\right) \\ ubound(p) &= \left(\frac{b}{a_1}\right) p + \left(\frac{b - a_0}{a_1}\right) \end{aligned}$$

If $A(s(i))$ is a enclosed in the loop

$$\begin{array}{l} \text{DO i} = l, u \\ \quad \cdots \text{A(s(i))} \cdots \\ \text{END DO} \end{array}$$

then the symbolic set must also account for the bounds l and u:

$$i \in [\lceil max(l, lbound(p)) \rceil : \lfloor min(u, ubound(p)) \rfloor]$$

Note that the loop bounds must be known at compile time (after constant propagation and induction variable substitution [5]). Figure 2a illustrates this set for the case of $l = 20$, $u = 480$, and an array $A(1 : 500)$ distributed by blocks (with $b = 63$) across eight processors. Figure 3a shows the same set for a blocked-cyclic distribution with a block size of $b = 10$.

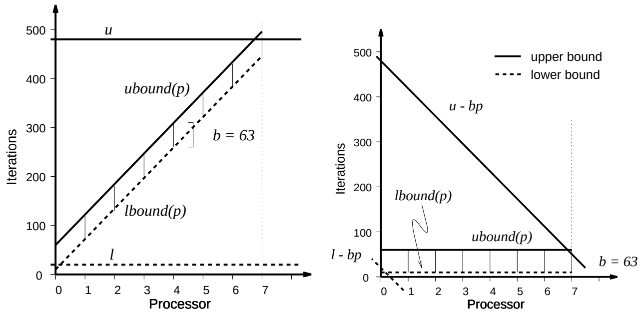
Symbolic Set Operations

To perform set operations on this representation, it is necessary to be able to perform comparisons between bound expressions. For linear expressions, comparisons can be made within a desired range to determine the relation between the two expressions. This linear comparison operation is extended to perform comparison of binary operations (*min*, *max*) of linear expressions. With symbolic

[a] Currently, only loop skips of 1 are allowed, but steps that are due to a non-unity coefficient of the loop index in the subscript expression are supported (see *Set Transformations*).

[b] floor and ceiling operations must be applied to the bounds only when examining the sets in the Global Iteration domain.

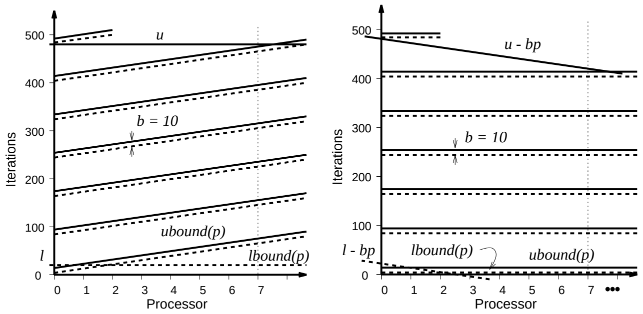
A(i) distributed by blocks of size 63



(a) Global Iterations (b) Local Iterations

Figure 2: Block Distribution

A(i) distributed cyclic with blocks of size 10



(a) Global Iterations (b) Local Iterations

Figure 3: Cyclic Distribution

comparisons defined between the available bound expressions, the following set operations are defined to operate on n-dimensional sets:

`is_subset`	true if A is a subset of B	$A \subseteq B$
`are_disjoint`	true if A and B are disjoint	$A \cap B = \emptyset$
`union`	n-D union of two sets	$A \cup B$
`intersection`	n-D intersection of two sets	$A \cap B$
`difference`	n-D difference of two sets	$A - B$
`stretch`	*stretch* an n-D set A over B	
`location`	n-D *location* of B wrt A	

Other than the basic set operations, two new ideas have also been introduced: *stretch* and *location*.

Stretch Stretching of a set is performed by combining the bounds of one set with another to generate a list of sets that surround the area common to both sets. This is used with n 1-D difference operations to approximate an n-dimensional difference with the additional property of separating the resulting sets into the $3^n - 1$ neighbor regions that exist for an n dimensional set (see Figure 4).

Location The *location* of a set is the n-tuple of $\{-, none, +\}$ direction vectors. Given two sets, this computes the relative relationship that exists between them.

The use of the location and stretch operations (as well as the application of the other operations) will become apparent during the actual analysis of reference and communication patterns.

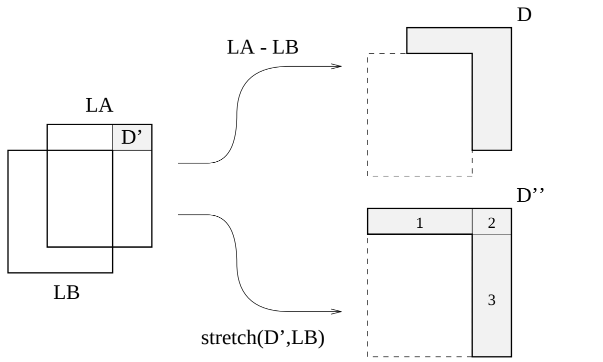


Figure 4: Comparison of Difference and Stretch

Set Transformations

During analysis of the reference patterns in a program, symbolic sets are transformed among different domains. Initially, the array references are collected directly from the serial program in terms of the *global iteration space* (see Section 2.2). Altogether, four separate domains can be defined: global and local iterations, global and local indices. Transformations (and their inverses) between the different domains are defined as:

$$\begin{array}{lccc} & \text{Global} & & \text{Local} \\ \text{Iterations} & \mathbf{GI} & \xrightarrow{\lambda_p(i)} & \mathbf{LI} \\ & s(i)\downarrow & & \downarrow s(i) \\ \text{Indices} & \mathbf{GN} & \xrightarrow[\tau_p(i)]{} & \mathbf{LN} \end{array}$$

Global Iterations to Global Indices The subscript function $s(i)$ of an array reference transforms points in the global iteration space to the global index space (see Figure 5). Currently, $s(i)$ is allowed to be a linear function on a loop variable i:

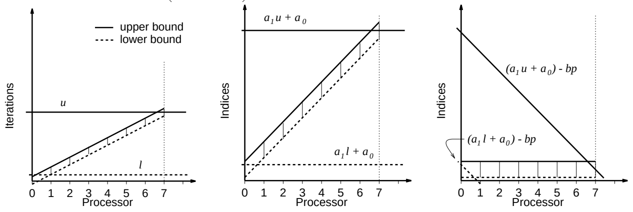
$$s(i) = a_1 i + a_0 \qquad (1)$$

To transform entire sets, the *step* must be scaled by a_1:

$$step' = a_1 \times step$$

Given a global iteration set $[u : v : w]$, the corresponding global index set is $[s(u) : s(v) : a_1 w]$.

A(2i + 5) distributed by blocks



Global Iterations Global Indices Local Indices

Figure 5: Linear Reference Translations

Global Indices to Local Indices The function $\tau_p(x)$ transforms points in the global index space to indices local to a processor p (see Figure 5):

$$\tau_p(x) = x - bp \tag{2}$$

where b is the block size of distribution. Hence, a global index set $[u : v : w]$ can be transformed into a local index set $[\tau_p(u) : \tau_p(v) : w]$.

Global Iterations to Local Iterations The function λ_p transforms points from the global iteration space to iterations local to a processor p (Figures 2b and 3b):

$$\lambda_p(i) = i - \frac{b}{a_1}p \tag{3}$$

2.2 Access Iteration Sets

Let R be an array reference enclosed in a loop, then the ACCESS set[(c)] of R with respect to a processor p, denoted $\text{ACCESS}(R, p)$, is the set of global iterations for which R accesses data owned by p. For example, for an array A, the ACCESS set of a reference $A(s(i))$ is:

$$\begin{aligned}&\text{ACCESS}(A(s(i)), p) = \\ &\quad [\lceil max(l, s^{-1}(1 + bp))\rceil : \lfloor min(u, s^{-1}(b + bp))\rfloor]\end{aligned} \tag{4}$$

where i is the loop variable, l and u are the lower and upper loop bounds (loop skip is 1), b is the distributed block size of A, and $s(i)$ is the subscript function of the array reference.

The same concept can be extended for nested loops and multidimensional arrays by defining a subscript expression for each dimension as a function of the loop variables.

2.3 Computation Partitioning

An efficient implementation of the owner computes rule must avoid the runtime overhead of computing ownership. For computations inside a loop nest, this is done by reducing the loop bounds according to the Reduced Iteration Set [1] (RIS) of the loop, which is the union of the ACCESS sets of the *lhs* references in the loop with respect to a processor p. The RIS represents the largest subset of the iteration space for which p does some work in *every* iteration. Since the RIS is in the global iteration space, if used directly, a translation τ_p is required for the reference at every iteration, which is very costly:

```
DO i ∈ RIS
    A(τp(s(i))) = ···
END DO
```

A better approach is to apply λ_p (Equation 3) to RIS to obtain the *Local* RIS (LRIS):

$$LRIS = \lambda_p(RIS)$$

A loop using the LRIS is much more efficient:

```
DO i ∈ LRIS
    A(s(i)) = ···
END DO
```

(c) ACCESS sets are called *Local Iteration Sets* in [1] which causes confusion since these sets describe global iterations.

If the ACCESS set of a particular *lhs* in the loop is a *proper* subset of the RIS, then the corresponding statement must be *masked* [1] to make its execution conditional. Optimizations such as *mask merging* (consecutive assignment statements sharing the same mask) and *mask extraction* (multiple occurrences of a mask inside a loop are coalesced and extracted out of the loop) are performed automatically by the compiler [8].

2.4 Communication Analysis

To generate communication for regular computations, we can also apply symbolic set operations on the ACCESS sets of both the *lhs* and *rhs* references of assignment statements. This creates communication sets that encompass information about whether communication is needed, and if so under which iterations, between which pairs of processors, and what subsections of which arrays to send or receive.

Receive/Send Iteration Sets

The first communication sets to compute are the RECEIVE and SEND sets. These sets indicate which iterations require communication of data. For each statement S,

$$S : lhs = \mathcal{F}(rhs_1, rhs_2, rhs_3, \ldots)$$

$\text{RECEIVE}(rhs_i, p)$ is the set of iterations for which the execution of S in processor p requires nonlocal data due to the reference rhs_i. Similarly, $\text{SEND}(rhs_i, p)$ is the set of iterations for which reference rhs_i is local to p, but lhs is not, so S is executed in a processor other than p [1].

For each *rhs* reference in each assignment statement, we must determine their RECEIVE and SEND sets. From the owner computes rule, the RECEIVE set is[(d)]

$$\text{RECEIVE}(rhs, p) = \text{ACCESS}(lhs, p) - \text{ACCESS}(rhs, p) \tag{5}$$

In other words, it is the set of iterations for which *lhs* is local (and hence p executes S) but *rhs* is not available and must be *received* from another processor.

Similarly, the SEND set is given by

$$\text{SEND}(rhs, p) = \text{ACCESS}(rhs, p) - \text{ACCESS}(lhs, p) \tag{6}$$

As before, equations 5 and 6 can be extended to multidimensional arrays and meshes by maintaining one expression for each dimension.

In/Out Index Sets

To actually generate communication, the array subsections to be communicated must be known. The RECEIVE and SEND sets have to be translated from the *global iteration* space to the *local index* space of a processor p. Applying Equations 1 and 2, the IN and OUT local index sets are obtained from the RECEIVE and SEND iteration sets:

$$\text{IN}(A(s(i)), p) = \tau_p(s(\text{RECEIVE}(A(s(i)), p))) \tag{7}$$

$$\text{OUT}(A(s(i)), p) = \tau_p(s(\text{SEND}(A(s(i)), p))) \tag{8}$$

These sets can also be extended for multidimensional cases as before.

(d) Where "$-$" denotes either a set difference or a *stretch* operation as previously defined in Section 2.1.

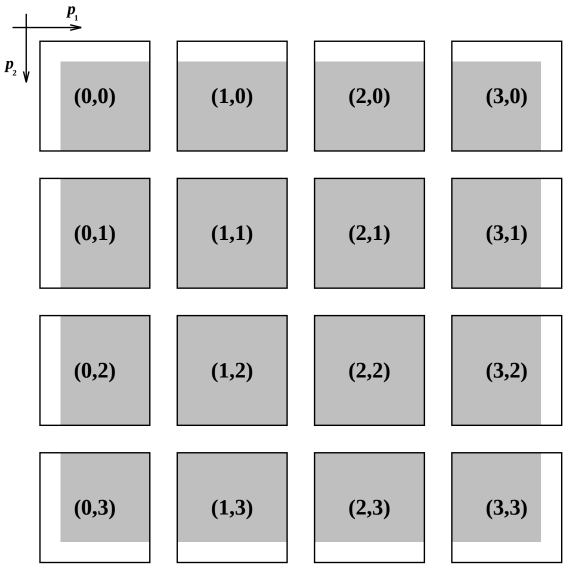


Figure 6: Iteration Space for Jacobi

To/From Processor Sets

Both absolute and relative processor addressing is supported in the compiler. In general, computation of TO and FROM processor sets is addressed in the synthesis of high-level communication [9]. For nearest-neighbor communication, however, knowing the relative offset of either of the IN or OUT sets is sufficient to determine processor locations in the mesh. The *location* set operation is used to map index set offsets to the relative processor addresses.

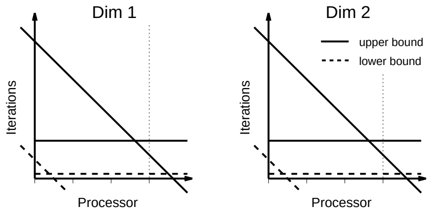
2.5 Example: Jacobi's Iterative Method

We conclude this section with an illustration of some of the main ideas using Jacobi's Iterative method as an example (see Figure 7). The A and B arrays are distributed by blocks in both dimensions on a 2-dimensional processor mesh. The shaded region in Figure 6 represents the global iteration space of the loop nest. In the figure, each processor is addressed by a pair (p_1, p_2) which indicates its position in the processor mesh. The sets which describe the local RIS for each dimension are shown in Figure 8. Figure 9 shows sections of the array B that require communication determined by computing the IN and OUT sets (shown in Figure 10). Note that overlap regions are necessary for the communication described by these sets (this will be discussed in Section 3.2).

```
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
```

Figure 7: Serial version of Jacobi's Iterative Method



$$\text{LRIS}_i(p_1) = [max(l_1 - b_1 p_1, 1) : min(u_1 - b_1 p_1, b_1)]$$
$$\text{LRIS}_j(p_2) = [max(l_2 - b_2 p_2, 1) : min(u_2 - b_2 p_2, b_2)]$$

Figure 8: Local RIS for i and j Loops in Jacobi

3. COMMUNICATION GENERATION

3.1 Communication Optimizations

Once the IN and OUT index sets have been computed, higher level decisions can be made to determine where to place the necessary communication. To reduce communication overhead, many optimization techniques can be applied during communication generation. Currently, these optimizations include message coalescing, vectorization, and aggregation [1]. Data dependence information is used to determine whether a given communication optimization is applicable, as well as exactly where the communication should be placed. Three lists of *communication descriptors* [8] (structures containing the IN/OUT and TO/FROM sets) are maintained for each loop header:

- BEFORE - before entering the loop
- TOP - at the start of each loop iteration
- BOTTOM - at the end of each loop iteration

These placements allow communication to be placed in the appropriate location with respect to any dependencies that may exist. If a reference is independent on a given loop variable, then its descriptor could be placed in the BEFORE list so that it would be performed only once over the entire loop. If a cross-iteration dependence exists then pipelining can be employed by splitting the send/receive operation between the TOP and BOTTOM lists.

Message Coalescing When statically analyzing the access patterns, redundant communications can be eliminated and the data can be reused from previous communication operations. By performing a union on overlapping index sets during placement of the communication descriptors, each element will be communicated at most once.

Message Vectorization Message vectorization combines element messages within a loop nest into larger *vectorized* groups of elements in a single message. By checking each loop nest from the innermost outward, the vectorization level is selected as the outermost nesting level that

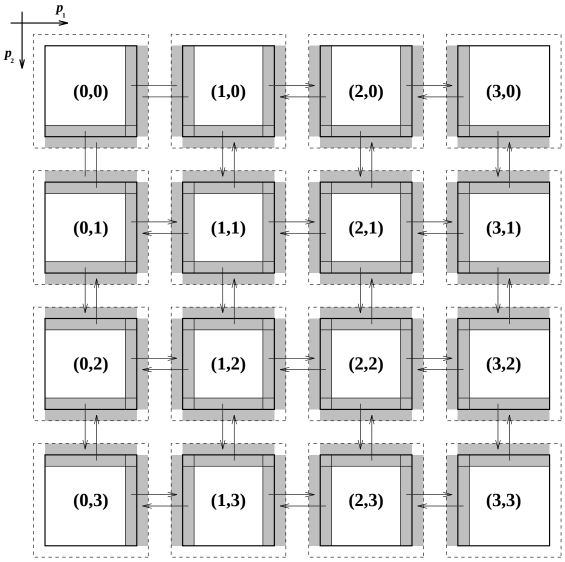
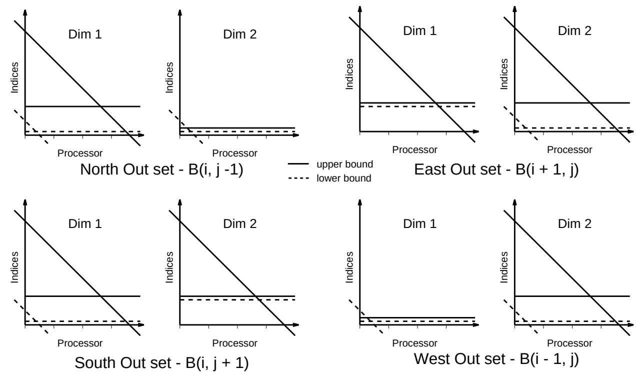


Figure 9: Communication with 4 Neighbors



$$\mathrm{Out}(B(i,j-1),p_1) = [max(l_1 - b_1p_1, 1) : min(u_1 - b_1p_1, b_1)]$$
$$\mathrm{Out}(B(i,j-1),p_2) = [max(l_2 - b_1p_2, 1) : 1]$$

$$\mathrm{In}(B(i,j-1),p_1) = [max(l_1 - b_1p_1, 1) : min(u_1 - b_1p_1, b_1)]$$
$$\mathrm{In}(B(i,j-1),p_2) = [max(l_2 - b_2p_2, b_2 + 1) : b_2 + 1]$$

(expressions for *North* In/Out sets)

Figure 10: Communication Index Sets

does not contain any cross iteration dependence for the reference. The "itemwise" messages are *vectorized* as they are lifted out of the loop nests until they are placed on the communication descriptor at the vectorization level.

Coalescing vectorized messages now implies that any overlapping *regions* between messages are only communicated once. With symbolic sets, unions can be performed among entire vectorized messages to eliminate the redundant communication of the common regions.

Message Aggregation By grouping multiple messages with the same destination into a single message, the frequency of communication operations is further reduced. Communication operations which are placed at the same communication descriptor are first sorted by their destination. Messages being sent to the same destination can then be collected into a single communication operation.

3.2 Communication Summary Nodes

A *summary node* is constructed for each communication descriptor to facilitate the generation of communication. The index sets from the descriptors are analyzed to record the layout and length of the message for each dimension of the array. The mapping of array dimensions to the mesh is also recorded so that the array sections can be correlated with the mesh dimensions. Block distributed arrays can now be reduced, requiring analysis of the summary nodes for possible overlap regions.[e] Once overlap regions have been set, communication operations (along with possible packing operations) can now be generated from the summarized bound information. Finally, communication masks can be generated directly using the To and From sets present in the communication descriptor.

[e] If cyclic distributed arrays were reduced, overlap regions could potentially exist between any of the sub-blocks, not simply at the boundaries, requiring further (complex) index translation.

Overlap Regions

Overlap regions need to be computed before **any** communication or buffering operations are generated. The extension of the bounds of an array directly affects the layout of the data and, therefore, whether or not packing needs to be performed with the communication.

To compute the overlap region for an array, the bounds of all corresponding In index sets are examined. Since the array bounds in the symbol table may have already been reduced, direct comparison of these bounds with the target destination of the In set will reveal the need for overlap areas. For each communication operation in the program, a destination region will be computed. If the symbol table entry for that reference is not large enough to enclose the region, the bounds are expanded for that entry. The need for overlap regions can be seen in Figure 9.

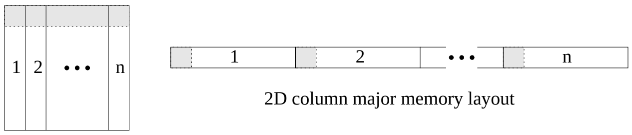
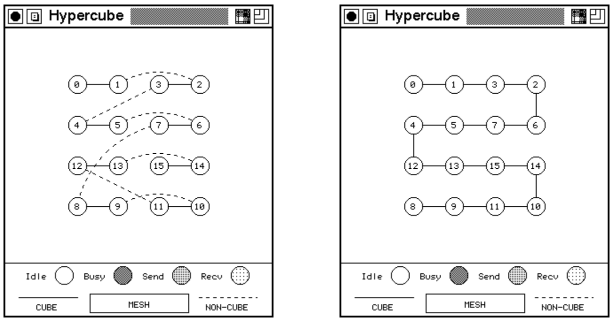


Figure 11: Buffering of a Non-Contiguous Region

Buffering Determination

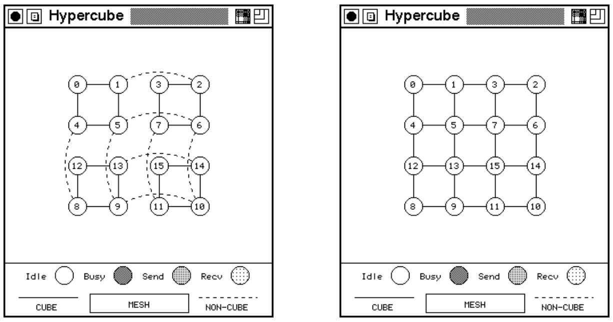
It is also possible to detect the need for message packing by examining the communication summary nodes. Messages must be contiguous when they are sent and as they are received. In Figure 11, a two-dimensional array is shown with a column-major data layout (used in FORTRAN). It should be noted that the elements of a row in the array are located in memory such that each are separated by a distance equal to the column length. If a single row of the array were to be communicated, a contiguous intermediate buffer would be required.

The dimensions of the section to be communicated are analyzed in the order specified by its memory layout. For



(a) Direct Mapping (b) Perfect Mapping

Figure 12: Mapping of a 1×16 Array



(a) Direct Mapping (b) Perfect Mapping

Figure 13: Mapping of a 4×4 Mesh

any given dimension, if the section does not cover the entire range of that dimension and if the next dimension in the layout accesses more than a single element (more than a unit-wide slice), then the entire section will have to be packed. Packing is also required when aggregating sections from different references into a single message regardless of their individual layouts.

3.3 Library Support

Library support is used to maintain machine independence (both for communication and mapping purposes) and to provide the pack and unpack operations. Also, by supporting certain communication libraries, performance evaluation through execution tracing is also made available.

Communication Library Support

Support for specific machines is provided through a generic library interface. For each supported communication library, abstract functions are mapped to the corresponding library-specific routines at compile-time. Since more information is carried internally than is required by any specific implementation, the mapping is obtained by selecting a subset of the available parameters. Both the communication operations, as well as a group of setup and cleanup statements are defined for each library.

Currently, library interfaces have been implemented for the Intel iPSC communication library [15], the Parasoft Express communication library [16], and the Portable Instrumented Communication Library (PICL) [17, 18]. Both Express and PICL provide execution tracing and support many different machines as well.

Processor Mapping

Internally, the compiler performs all analysis and communication generation for a virtual mesh topology. At runtime, this virtual mesh is mapped to the physical machine through the use of mapping functions. These functions are implemented for each target machine to efficiently map the mesh to the target architecture, maintaining the nearest neighbors of the virtual mesh as nearest neighbors on the machine whenever possible [19]. Processor mapping functions include:

- `gridinit(dim, num, nproc)` - Initialize mapping of a mesh of `dim` dimensions and `num[dim]` processors in each dimension (with `nproc` total available processors)
- `gridcoord(proc, coord)` - Given a processor ID, return the coordinates in the current mesh configuration
- `gridproc(coord)` - Given the processor mesh coordinates, return the processor ID
- `gridrel(coord, dim, rel)` - Given the processor coordinates, a mesh dimension, and a relative offset, return the processor ID of the relative node
- `gridrel`n`(coord, rel`$_1$`, ...rel`$_n$`)` - Given a processor number and a relative offset for **each** of n mesh dimensions, return the processor ID

If the target machine has a hypercube topology, then it is possible to map a D-dimensional mesh onto it while maintaining **all** nearest-neighbor links as physical links. Even though communication latency is not much different for any two given nodes, mapping the mesh in nearest-neighbor fashion is important when communication rates are high. Contention will become a problem if communication is not distributed evenly (which is more likely if the mapping is **not** by nearest neighbor).

Figure 12 shows a direct mapping of a 16×1 mesh on a 16-node hypercube (which results in many non-neighbor links), and a *perfect* Gray code mapping (which preserves all physical neighbor links) Similarly, Figure 13 shows the direct and perfect mappings for a 4×4 mesh.

Communication Buffering

To facilitate the packing and unpacking of data for communication, two routines are also defined for each cardinality of arrays that may need packing:

- `pack`$_n$`(a, byte, lb`$_1$`, ub`$_1$`, ...lb`$_n$`, ub`$_n$`,`
  `ls`$_1$`, us`$_1$`, sk`$_1$`, ...ls`$_n$`, us`$_n$`, sk`$_n$`, buf)`
- `unpack`$_n$`(a, byte, lb`$_1$`, ub`$_1$`, ...lb`$_n$`, ub`$_n$`,`
  `ls`$_1$`, us`$_1$`, sk`$_1$`, ...ls`$_n$`, us`$_n$`, sk`$_n$`, buf)`

These functions are implemented separately for each of the possible degrees since the dimensions of arrays found in programs are very limited and a generalized pack/unpack would not be as efficient. The layout is specified by the size of the elements, the lower and upper bounds of the array (`lb`$_i$`, ub`$_i$), as well as the lower and upper bounds describing the section to pack (`ls`$_i$`, us`$_i$) for each dimension. A skip factor is also provided for each section description to support messages which require a stride between elements.

Library support can be seen in the resulting SPMD parallel program (see Figure 15 at the end of this paper) compiled from the serial version shown earlier in Figure 7.

Table 1: Performance Summary

Test	N	Virtual Grid	Time (s)	Speedup	Efficiency
ADI	1	$1 \times 1 \times 1$	2.780	1.000	1.000
	2	$1 \times 2 \times 1$	1.369	*2.031*	*1.016*
	4	$1 \times 4 \times 1$	0.695	*4.002*	*1.001*
	8	$1 \times 8 \times 1$	0.349	7.968	0.996
	16	$1 \times 16 \times 1$	0.180	15.441	0.965
Expl	1	1×1	14.511	1.000	1.000
	2	2×1	7.580	1.914	0.957
	4	4×1	3.777	3.842	0.961
	8	8×1	1.902	7.629	0.954
	16	16×1	0.982	14.772	0.923
Jacobi	1	1×1	83.954	1.000	1.000
	2	2×1	37.566	*2.235*	*1.118*
	4	4×1	18.958	*4.428*	*1.107*
	8	4×2	10.250	*8.191*	*1.024*
	16	4×4	5.165	*16.253*	*1.016*

4. RESULTS

A group of small scientific program kernels [1, 20] are used to examine the performance of the output of the PARADIGM compiler on a real machine. The selected program fragments include an alternating direction implicit (ADI) integration (Livermore kernel 8), a 2-D explicit hydrodynamics routine (Livermore kernel 18), and Jacobi's iterative method. Array dimensions and loop bounds are determined at compile time with the constant propagation and induction variable passes in the compiler [5]. Both the virtual mesh configuration as well as the data partitioning of the arrays are automatically selected by the compiler [6] before generation of the SPMD program. The dimensions of the major arrays present in the programs are:

- ADI Integration - 1024 and $4 \times 1024 \times 2$ element arrays
- Explicit Hydrodynamics - 1024×7 element arrays
- Jacobi's Iterative Method - 500×500 element arrays

Using all optimizations, each of the test programs are compiled for an Intel iPSC/2 to examine the resulting performance over a range of processors. In Table 1, each program's partitioning configuration is shown along with the resulting execution time, speedup and efficiency. Superlinear speedup (due to cache effects associated with reduced array bounds and smaller local working sets) is italicized in the table. Both the execution time and speedup of the test programs are also shown in Figure 14.

Table 1 shows that both the ADI and Explicit Hydrodynamics programs have linear distributions for all machine sizes. This is due to the fact that the computation performed in both of these programs is mainly in a single dimension of the data arrays. The aligning capability of the distribution module of the compiler can correctly extract this information even though different array dimensions are used in different references to perform the computation.

It is interesting to notice in Table 1 that the Jacobi's iterative method has enough parallelism in two dimensions of the computation to allow for two-dimensional partitionings. Since communication becomes less expensive for 2-D partitionings for larger mesh sizes, the compiler has more freedom in selecting the mesh configuration for larger machines [14].

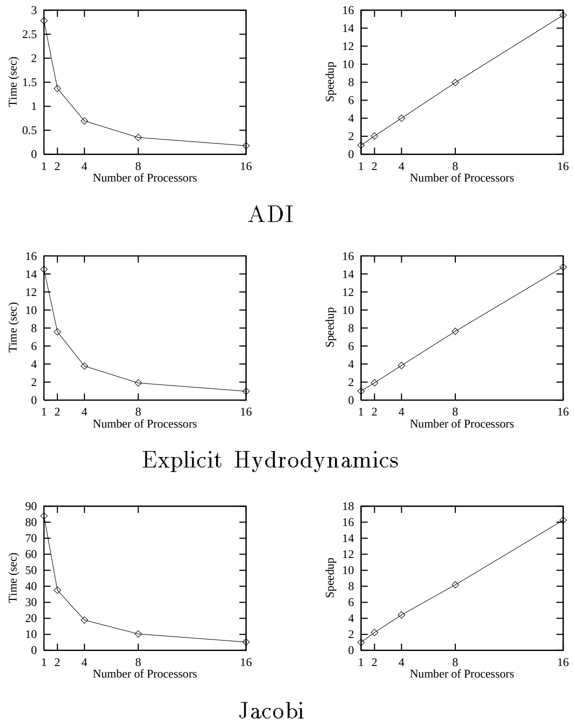


Figure 14: Performance of Test Programs

5. CONCLUSIONS

The purpose of this work is to automatically parallelize serial programs, efficiently mapping the abstract model to a given target machine. This is accomplished through the use of multidimensional partitioning schemes and processor mapping functions. Symbolic sets are heavily used to provide a uniform means of describing machine-wide access and communication patterns. Also, to provide machine independence, both the data distribution module as well as the library interface were developed such that the compiler could be easily retargeted.

Given the high efficiencies of the test programs, the applicability of this technique to regular computations is easily seen. Exactly how useful the techniques presented in this paper can be depends directly on the class of input programs that are to be parallelized. The current state of the compiler provides a useful testbed to develop further compiler optimizations for parallelization on distributed-memory multicomputers. For programs which show unbalanced computations, cyclic distributions will be added to the compiler to allow better load distributions. Research is also currently underway to support multithreaded execution [13] as well as to take advantage of functional parallelism [12] to address other classes of programs.

In conclusion, the two most complex tasks facing a user in parallelizing serial programs for a distributed-memory multicomputer are the distribution of the data and the implementation of interprocessor communication. In this paper, it has been shown that both of these tasks can be handled by the compiler for regular computations, thereby relieving the programmer of the burden of parallelization.

Acknowledgements: The authors would like to thank the referees for their helpful input. We would also like to thank Christy Palermo for her help and suggestions with the development of the symbolic comparison algorithms which are used in the implementation of symbolic sets. We would also like to thank Manish Gupta for his work, and many helpful discussions, on automatic data partitioning.

REFERENCES

[1] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiler Support for Machine-Independent Parallel Programming in Fortran D," Tech. Rep. TR90-149, Rice University, February 1991.

[2] C. Koelbel, "Compile-Time Generation of Regular Communications Patterns," in *Proceedings of 1991 ACM International Conference on Supercomputing*, (Albuquerque, NM), November 1991.

[3] C. Koelbel and P. Mehrotra, "Compiling Global Name-Space Parallel Loops for Distributed Execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 440–451, October 1991.

[4] H. Zima, H. Bast, and M. Gerndt, "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing*, vol. 6, pp. 1–18, 1988.

[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 1989 International Conference on Parallel Processing*, pp. II,39–48, August 1989.

[6] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 179–193, March 1992.

[7] M. Gupta and P. Banerjee, "Compile-Time Estimation of Communication Costs on Multicomputers," in *Proceedings of 6th International Parallel Processing Symposium*, (Beverly Hills, CA), March 1992.

[8] E. Su, "Automating Parallelization of Regular Computations for Distributed-Memory Multicomputers," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1993.

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

[10] K. P. Belkhale and P. Banerjee, "Approximate Algorithms for the Partitionable Independent Task Scheduling Problem," in *Proceedings of the 1990 International Conference on Parallel Processing*, pp. 72–75, August 1990.

[11] K. P. Belkhale and P. Banerjee, "A Scheduling Algorithm for Parallelizable Dependent Tasks," in *International Parallel Processing Symposium*, pp. 500–506, 1991.

[12] S. Ramaswamy and P. Banerjee, "Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers," *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993.

[13] A. Lain, J. G. Holm, and P. Banerjee, "Compiler Transformations for Multithreading and Message-Driven Computation," submitted to the *1993 ACM International Conference on Supercomputing*, 1993.

[14] D. J. Palermo, E. Su, and P. Banerjee, "Evaluation of Communication Optimizations used in the PARADIGM Compiler," submitted to the *1993 ACM International Conference on Supercomputing*, 1993.

[15] Intel Corporation, *iPSC/2 and iPSC/860 User's Guide*, June 1990.

[16] Parasoft Corporation, Pasadena, CA, *Express Reference Guide for FORTRAN Programmers*, 1992.

[17] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "PICL: A Portable Instrumented Communication Library, C reference manual," Tech. Rep. ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[18] M. T. Heath and J. A. Etheridge, "Visualizing the performance of parallel programs," *IEEE Software*, vol. 8, pp. 29–39, September 1991.

[19] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[20] F. McMahon, "The Livermore Fortran Kernels: A computer test of the numerical performance range," Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, 1986.

```
program jacobi
character m$buf(500)
integer m$numdim, m$num(2), m$to(-1:1,-1:1), p(2)
real A(125, 125), B(0:126, 0:126)

m$numdim = 2   {number of mesh dimensions}
m$num(1) = 4   {number of processors on mesh dim. 1}
m$num(2) = 4   {number of processors on mesh dim. 2}
call m$gridinit(m$numdim,m$num,numnodes())
call m$gridcoord(mynode(),p)

m$to(-1,0) = m$gridrel2(p,-1,0)
m$to(0,-1) = m$gridrel2(p,0,-1)
m$to(0,1) = m$gridrel2(p,0,1)
m$to(1,0) = m$gridrel2(p,1,0)

do k = 1, 10
   if (p(2) .ge. 1) then
      call csend(0,b(1,1),500,m$to(0,-1),1)
   end if
   if (p(2) .le. 2) then
      call crecv(0,b(1,126),500)
   end if
   if (p(2) .le. 2) then
      call csend(1,b(1,125),500,m$to(0,1),1)
   end if
   if (p(2) .ge. 1) then
      call crecv(1,b(1,0),500)
   end if
   if (p(1) .ge. 1) then
      call f$pack2(b,4,0,126,0,126,1,1,1,1,125,1,m$buf)
      call csend(2,m$buf,500,m$to(-1,0),1)
   end if
   if (p(1) .le. 2) then
      call crecv(2,m$buf,500)
      call f$unpack2(b,4,0,126,0,126,126,126,1,1,125,1,m$buf)
   end if
   if (p(1) .le. 2) then
      call f$pack2(b,4,0,126,0,126,125,125,1,1,125,1,m$buf)
      call csend(3,m$buf,500,m$to(1,0),1)
   end if
   if (p(1) .ge. 1) then
      call crecv(3,m$buf,500)
      call f$unpack2(b,4,0,126,0,126,0,0,1,1,125,1,m$buf)
   end if
   do j = max(2 - 125 * p(2), 1), min(499 - 125 * p(2), 125)
      do i = max(2 - 125 * p(1), 1), min(499 - 125 * p(1), 125)
         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 - 125 * p(2), 1), min(499 - 125 * p(2), 125)
      do i = max(2 - 125 * p(1), 1), min(499 - 125 * p(1), 125)
         B(i,j) = A(i,j)
      end do
   end do
end do
end
```

Figure 15: SPMD version of Jacobi's Iterative Method