# Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers*

Ernesto Su, Daniel J. Palermo, and Prithviraj Banerjee

Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 West Main Street, Urbana, IL 61801, U.S.A.

**Abstract:** The computation partitioning, communication analysis, and optimization phases performed during compilation for distributed-memory multicomputers require an efficient way of describing distributed sets of iterations and regions of data. Processor Tagged Descriptors (PTDs) provide these capabilities through a single set representation parameterized by the processor location for each dimension of a virtual mesh. A uniform representation is maintained 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 the effect on all processors in a dimension can be captured in a single symbolic operation. In addition, PTDs 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. Using the symbolic features of PTDs it is also possible to generate code for variable numbers of processors, thereby allowing a compiled program to run unchanged on varying sized machines. The PARADIGM (PARAllelizing compiler for DIstributed-memory General-purpose Multicomputers) project at the University of Illinois utilizes PTDs to provide an automated means to parallelize serial programs for execution on distributed-memory multicomputers.

**Keyword Codes:** C.1.2; E.2
**Keywords:** Multiprocessors; Data Storage Representations

## 1 Introduction

A parallelizing compiler for distributed-memory multicomputers must be able to: (1) describe decompositions of arrays and loops across a number of processors, and (2) translate indices and iterations between global and local address and iteration spaces. There is a need for a representation capable of describing *partitioned* polyhedra of arbitrary dimensions in any of these spaces. It must also provide a flexible structure for performing high-level set operations required in both computation partitioning and communication optimizations [11]. This paper describes how PTDs efficiently provide all of these capabilities. In addition, a key feature of PTDs is the support of code generation for variable numbers of processors. Not only does this allow a compiled program to run unchanged on

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

varying sized machines, but it is also useful when selecting the number of tasks for multi-threaded execution [9] or for varying the number of processors assigned to a given task when utilizing functional parallelism [12]. In contrast, most ongoing work on compilers for distributed memory [3, 5, 8, 10] can only compile for a fixed number of processors.

Earlier structures describing index or iteration space arose in the context of interprocedural data dependence analysis. They include the *Regular Section Descriptor* (RSD) [4] and the *Data Access Descriptor* (DAD) [2]. RSDs can only describe a single row, column, or diagonal of an array, or the entire array. DADs describe convex polyhedra bounded by hyperplanes that are either orthogonal to an axis or at 45° angles with a pair of axes. Both representations are only able to define regions in an unpartitioned address space. The Fortran D compiler [8] uses the RSD augmented with three component sections for each dimension to handle boundary conditions, which are often present in partitions owned by boundary processors. However, when the exact set boundaries are not computable at compile time, it maintains an individual iteration or index set per processor.

The Stanford SUIF compiler system [15] models data and computation decompositions using a form of parametric integer programming. Its representation is parameterized with processor information and is able to handle symbolic block sizes but results in multi-version copies of loops.[1] It also only supports global index and iteration domains, while local index translation is performed at run time for all distributed array accesses [1].

The PTD and defined operations are presented in Section 2. The application of PTDs in the PARADIGM compiler is described in Section 3, and conclusions appear in Section 4.

# 2 Processor Tagged Descriptors

The PTD is a set representation parameterized by processor coordinates. This provides a uniform and efficient means of describing sets of iterations or sections of arrays that are partitioned across processors. A PTD in $n$ dimensions is a collection of one-dimensional $\mathrm{PTD}_i$ $(1 \leq i \leq n)$, each of which has as a lower and upper bound. Each bound is a symbolic expression involving a processor coordinate $(p_i)$ along dimension $i$ of the mesh:

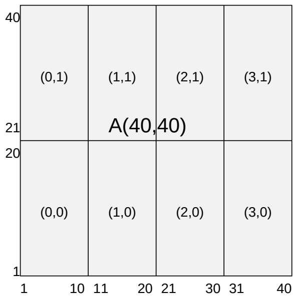
$$\begin{aligned} PTD_i &= [lower\_bound : upper\_bound] \\ lower\_bound &= bound \mid \lceil bound \rceil \\ upper\_bound &= bound \mid \lfloor bound \rfloor \end{aligned} \qquad bound = \begin{cases} linear\_expr(p_i) \\ min(bound, bound) \\ max(bound, bound) \end{cases}$$

$$linear\_expr(p_i) = sym\_expr \cdot p_i + sym\_expr$$

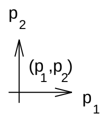
where *sym_expr* is a scalar symbolic expression (expressions involving constants and program variables). Each $\mathrm{PTD}_i$ captures the range of integers between the lower and upper bounds in a single dimension. Together, they span the entire $n$-dimensional region. Since the PTD is parameterized by processor coordinates, it provides a uniform representation for every processor in the mesh. This is regardless of the shape and size of the partition and whether it resides in a boundary or an interior processor in the mesh. Furthermore, set operations on these partitions are very efficient because all processors in one or more dimensions can be captured into a single symbolic set operation.

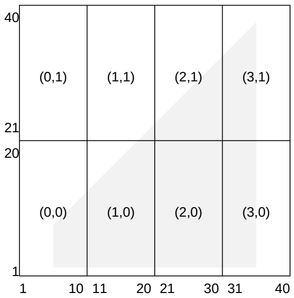
Figure 1 shows the PTD for an array $A(40, 40)$ partitioned by blocks on a $4 \times 2$ processor mesh. $A_1$ (first dimension of $A$) is distributed along the first dimension (4 processors) of the mesh and $A_2$ along the second dimension (2 processors). Section 3.1 defines this as the IMAGE of $A$. Each processor is identified by its coordinates $(p_1, p_2)$, with $p_1 \in \{0, 1, 2, 3\}$

[1] For each dimension of a decomposition, SUIF generates three different versions of each partitionable loop (two boundaries and a core computation). For $n$-dimensions this results in code expansion of $3^n$.




DO i = $5, 35$
  DO j = $2, i + 2$
    $\cdots A(i, j) \cdots$




$\text{PTD}_1 = [1 + 10p_1 : 10 + 10p_1]$
$\text{PTD}_2 = [1 + 20p_2 : 20 + 20p_2]$

$i \in \text{PTD}_1 = [max(5, 1 + 10p_1) : min(35, 10 + 10p_1)]$
$j \in \text{PTD}_2 = [max(2, 1 + 20p_2) : min(i + 2, 20 + 20p_2)]$

Figure 1: Partitioned Array $A(40, 40)$

Figure 2: Partitioned Iteration Space

and $p_2 \in \{0, 1\}$. Given any processor $p = (p_1, p_2)$, the bound expressions of the PTD shown in the figure define the regions of $A$ stored in $p$. Let $A(i, j)$ be referenced in the loop nest as shown above. If the iteration space were partitioned such that $p$ executes iteration $(i, j)$ if $p$ accesses $A(i, j)$ locally, the resulting description would be as shown in Figure 2. Although all of these regions have different sizes (from $\emptyset$ in $p = (0, 1)$ to $10 \times 19$ in $p = (2, 0)$) or shapes (null, rectangular, triangular, and trapezoidal), a single PTD describes them all. Section 3.1 refers to each partition as the Access iteration set of $A(i, j)$ with respect to $p$.

## 2.1 Variable Number of Processors

On most multicomputers, users are able to request the size of a machine partition on which to run programs. Given a fixed size array, the more processors available at run time, the smaller the array partition accessed by each processor (since $b_i \propto \frac{1}{P_i}$). The compiled program must adapt itself to the configuration of the machine partition on which it runs. The PTD supports variable processor compilation by retaining the block sizes of distributed array dimensions, $b_i$, as symbolic variables (described in Section 2.3).

Many implementation issues arise when dynamically allocating memory which has to be accessed in a multidimensional fashion. One solution is to *linearize* all partitioned $n$-D arrays, as used by APR in the FORGE xHPF parallelizer, at the expense of evaluating complicated array subscripts for each reference at run time. A simpler approach, currently used in PARADIGM, is to require the user to specify a *minimum processor configuration* at compile time. This allows memory to be statically partitioned for the minimum configuration (retaining any dimension information) while allowing the number of processors to grow beyond this lower bound. Since the required block sizes of the partitioned arrays decrease with increasing number of processors, all accesses are contained within the partitioning of the minimum configuration while retaining multidimensional access.

## 2.2 Set Operations

To work with PTDs, several relational tests are defined to detect subset, disjoint, and null set conditions. Union, intersection, and difference functions are also defined to perform multidimensional operations on the sets. Figure 3 shows algorithms for the PTD set operations. Note that the representation is closed under *intersection* while a *union*

is_subset$_{(1d)}(A, B)$
  **if** $A = \emptyset$ **return(true)**
  **else if** $B = \emptyset$ **return(false)**
  **else if** $((lb(A) \geq lb(B))$ **or** $(ub(A) \leq ub(B)))$
    **return(true)**
  **else return(false)**

(a) Subset Relation $(A_i \subseteq B_i)$

are_disjoint$_{(1d)}(A, B)$
  **if** $(A = \emptyset$ **or** $B = \emptyset)$
    **return(true)**
  **else if** $((ub(A) < lb(B))$ **or** $(lb(A) > ub(B)))$
    **return(true)**
  **else return(false)**

(b) Disjoint Relation $(A_i \bigcap B_i = 0)$

$(A = \emptyset) \Rightarrow (lb(A_i) > ub(A_i))$ $\triangleright$ for some dimension $i$

(c) Null Set Relation $(A = \emptyset)$

union$(A, B)$
  **for** $i = 1$ **to** $dim$ **do**
    **if** $A_i = \emptyset$ $A_i \bigcup B_i = B_i$
    **else if** $B_i = \emptyset$ $A_i \bigcup B_i = A_i$
    **else if** are_disjoint$_{(1d)}(A_i, B_i)$
      $A_i \bigcup B_i = A_i + B_i$
    **else** $A_i \bigcup B_i = [\min(lb(A_i), lb(B_i)) : \max(ub(A_i), ub(B_i))]$
  **done**

(d) Union Operation $(A \bigcup B)$

intersection$(A, B)$
  **for** $i = 1$ **to** $dim$ **do**
    **if** are_disjoint$_{(1d)}(A_i, B_i)$
      $A_i \bigcap B_i = \emptyset$
    **else** $A_i \bigcap B_i = [\max(lb(A_i), lb(B_i)) : \min(ub(A_i), ub(B_i))]$
  **done**

(e) Intersection Operation $(A \bigcap B)$

difference$(A, B)$
  **for** $i = 1$ **to** $dim$ **do**
    **if** are_disjoint$_{(1d)}(A_i, B_i)$ $A_i - B_i = A_i$
    **else**
      $T = [lb(A_i) : lb(B_i){-}1] \bigcup [ub(B_i){+}1 : ub(A_i)]$
      $kernel_i = T \bigcap A_i$
  **done**
  $A - B = \text{stretch}(kernel, A \bigcap B)$

(f) Difference Operation $(A - B)$

Figure 3: Algorithms for Performing Set Operations on PTDs

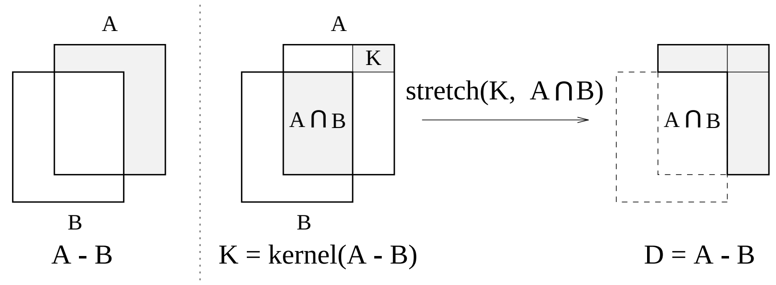
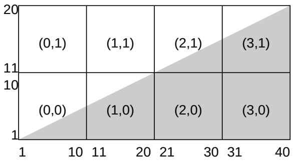
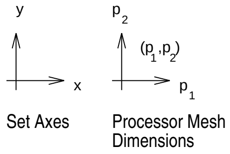


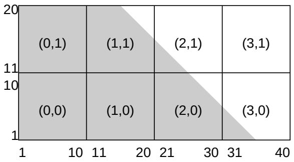

Figure 4: Stretch of a Difference Kernel



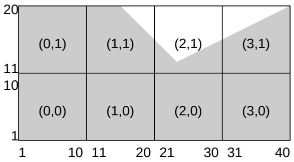

Partitioned Set $A$

$$x \in \mathrm{PTD}_1 = [1 + 10p_1 \;:\; 10 + 10p_1]$$
$$y \in \mathrm{PTD}_2 = [1 + 10p_2 \;:\; min(x/2, 10 + 10p_2)]$$



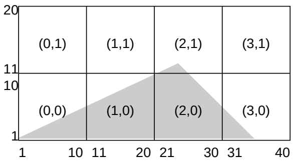
Partitioned Set $B$

$$x \in \mathrm{PTD}_1 = [1 + 10p_1 \;:\; min(35, 10 + 10p_1)]$$
$$y \in \mathrm{PTD}_2 = [1 + 10p_2 \;:\; min(35 - x, 10 + 10p_2)]$$



Partitioned Set Union $A \bigcup B$

$$x \in \mathrm{PTD}_1 = [1 + 10p_1 \;:\; 10 + 10p_1]$$
$$y \in \mathrm{PTD}_2 = [1 + 10p_2 \;:\; min(10 + 10p_2, max(x/2, 35 - x))]$$



Partitioned Set Intersection $A \bigcap B$

$$x \in \mathrm{PTD}_1 = [1 + 10p_1 \;:\; min(35, 10 + 10p_1)]$$
$$y \in \mathrm{PTD}_2 = [1 + 10p_2 \;:\; min(min(x/2, 10 + 10p_2), 35 - x)]$$

Partitioned Set Difference $A - B$

$$x \in \mathrm{PTD}_1 = [1 + 10p_1 \;:\; 10 + 10p_1]$$
$$y \in \mathrm{PTD}_2 = [min(36 - x, 11 + 10p_2) \;:\; min(x/2, 10 + 10p_2)]$$

Figure 5: Examples of Set Operations between two PTDs $A$ and $B$

or *difference* can potentially result in a list of sets. In Figure 3f, the *difference* operation is seen to perform a *stretch* operation after computing an $n$-dimensional "kernel". The kernel of a difference can be thought of as the region in which all 1-D differences intersect when projected into $n$ dimensions. The kernel is then *stretch*ed over the intersection of the original sets. This is done by choosing up to $(n-1)$ non-null dimensions from one set and selecting the remaining dimensions from the other (see Figure 4).

Figure 5 shows an example of how a union, intersection, and difference are performed between a triangular region $A$ and a trapezoidal region $B$, both partitioned on a $4 \times 2$ processor mesh. As shown in the figure, the first axis, $x$, is aligned with the first processor mesh dimension $p_1$, and $y$ is aligned with $p_2$. In the figure, each shaded region corresponds to the accompanying PTD expressions. $\mathrm{PTD}_1$ defines its $x$-values, and $\mathrm{PTD}_2$ defines the $y$-values as a function of the $x$-values (the expressions have been simplified where possible).

## 2.3 Symbolic Bound Comparison

Symbolic comparison is used to determine relations between bounds of PTDs as well as to maintain simplified bound expressions. The details involved in performing symbolic comparison between the expressions supported in PTDs are not included here due to space constraints. The basic approach taken to perform comparisons between two expressions is based upon a hierarchy of symbolic comparisons at different levels of complexity.

At the lowest level, symbolic comparison is performed between a pair of *symbolic scalar expressions* (expressions involving constants and variables) by simplifying a difference of the two expressions. For *symbolic linear expressions* (a linear expression with symbolic scalar coefficients), linear comparisons can be made within a desired range using scalar comparison between the coefficients. Comparison of *symbolic binary expressions* (a *min* or *max* of two symbolic linear expressions) can be performed with comparison lattices using linear comparisons between components. These levels are applied recursively to perform comparisons of nested binary operations on linear expressions of symbolic quantities.

**Symbolic Comparison with Variable Processors** Recall that the block sizes of partitioned arrays are allowed to vary with the number of processors $P_i$ assigned to the mesh dimension. Given a dimension of a distributed array of size $N$, the block size is computed as $b_i = \left\lceil \frac{N}{P_i} \right\rceil$. Since all array dimensions mapped to the same mesh dimension will vary together (with respect to $P_i$), comparison relations still exist between symbolic block sizes although the actual value of the block size is not known at compile time.

To support comparison of symbolic block sizes the *symbolic scalar comparison* operation is further abstracted using *mirrors*. A *mirror* retains information that would otherwise be lost if the source of the symbolic quantity were ignored. For symbolic scalars, an $n$-dimensional polynomial is used to *mirror* the value involving symbolic block sizes:

$$\beta_n P_n^{-1} + \beta_{n-1} P_{n-1}^{-1} + \ldots + \beta_2 P_2^{-1} + \beta_1 P_1^{-1} + c$$

where $\beta_i$ are block sizes based on a minimum processor configuration and $c$ is a constant. Whenever symbolic scalars are encountered, comparison can be performed by component-wise comparisons of each of the corresponding coefficients of the mirrors maintained for each scalar.[2] Any arithmetic operation that is performed on a symbolic expression must also be performed on the corresponding mirror polynomial. For use in the compiler, it is sufficient to support polynomial addition/subtraction, and scalar multiplication/division on these mirrors.

# 3 Use of PTDs in Compilation

Both data and computation decompositions can be represented using PTDs. In this section we examine the use of PTDs in the PARADIGM compiler [7]. During analysis of the reference patterns in a program, PTDs are transformed among different domains. Altogether, four domains can be defined: global and local indices (**GN** and **LN**), and global and local iterations (**GI** and **LI**). Transformation functions (and their inverses) among the different domains are:

$$\begin{array}{lccc} & \text{Global} & & \text{Local} \\ \text{Indices} & \mathbf{GN} & \xrightarrow{\tau_p(i)} & \mathbf{LN} \\ & s^{-1}(i) \downarrow\uparrow s(i) & & \uparrow s(i) \\ \text{Iterations} & \mathbf{GI} & \xrightarrow[\lambda_p(i)]{} & \mathbf{LI} \end{array}$$

[2]Since any dimension of a mirror is affected by the constant term, $c$ is added to non-zero coefficients during comparisons. Constant scalars also have implicit mirrors where $c$ is equal to the numeric value.

The subscript function $s(i)$ of an array reference $A(s(i))$, enclosed in a loop, transforms points from the loop's iteration space into the index space.[3] For simplicity, in this paper $s(i)$ is allowed to be a linear function[4] of a loop variable $i$: $s(i) = a_1 i + a_0$. The index translation function $\tau_p$ transforms points in the global index space into indices local to a processor $p$: $\tau_p(x) = x - bp$, where $b$ is the block size of distribution. Similarly, the iteration translation function $\lambda_p$ transforms points from the global iteration space into iterations local to a processor $p$: $\lambda_p(i) = i - \frac{b}{a_1}p$.

## 3.1 Data Partitioning

**IMAGE Global Index Sets** Using the data partitioning provided by the user or generated by the compiler [6], PTDs in the global index domain (**GN**) can be generated for each partitioned dimension of a distributed array. For each array $A(1 : N)$ distributed across a processor mesh dimension of $P$ processors indexed by $p$ $(0 \le p \le P-1)$, the corresponding PTD describing this partitioned array is

$$\mathrm{PTD} = [1 + bp : b + bp]$$

where $b = \left\lceil \frac{N}{P} \right\rceil$ is the *block size* of $A$. This PTD is called the IMAGE of $A$ on $p$, denoted $\textsc{Image}(A,p)$. This expression assumes that the alignment offset $\omega$ of $A$ on the mesh dimension is $\omega = 0$, and that the indices of $A$ start at 1. In general, we may have $A(\alpha : N + \alpha - 1)$ and $\omega \ne 0$, and then $\textsc{Image}(A,p)$ becomes

$$\textsc{Image}(A,p) = [max(\alpha, \alpha + \omega + bp) : min(N + \alpha - 1, b + \alpha - 1 + \omega + bp)]$$

If $\omega > 0$, then we need some $p < 0$ to address the leftmost $\omega$ elements of $A$. Similarly, if $\omega < 0$, then some $p > P - 1$ is required to address the rightmost $\omega$ elements of $A$. In either case, we can define $p' = p \ mod \ P$ as the real processor coordinate. To avoid unnecessarily obscuring the expressions derived, we assume $\omega = 0$ and $\alpha = 1$.

**ITERIMAGE and ACCESS Global Iteration Sets** From the IMAGE index set of $A$, the ITERIMAGE and ACCESS global iteration sets (in **GI**) of each reference $A(s(i))$ can be computed. The ITERIMAGE set of $A(s(i))$ with respect to a processor $p$ is the set of iterations $i$, unrestricted by loop bounds, such that $A(s(i))$ is stored in $p$. Since the partition of $A$ stored in $p$ is given by $\textsc{Image}(A,p)$, we can write

$$\begin{aligned} \textsc{IterImage}(A(s(i)),p) &= \{i \mid s(i) \in \textsc{Image}(A,p)\} \\ &= [\lceil s^{-1}(1+bp) \rceil : \lfloor s^{-1}(b+bp) \rfloor] \end{aligned}$$

For $A(s(i))$ enclosed in a loop nest in which the $i$-loop goes from $l$ to $u$, $\textsc{Access}(A(s(i)),p)$ is defined as the set of iterations $i$ in the $i$-loop for which $A(s(i))$ is stored in $p$:

$$\begin{aligned} \textsc{Access}(A(s(i)),p) &= \textsc{IterImage}(A(s(i)),p) \bigcap [l : u] \\ &= [\lceil max(l, s^{-1}(1+bp)) \rceil : \lfloor min(u, s^{-1}(b+bp)) \rfloor] \end{aligned}$$

[3] For simplicity of notation, it suffices to use one-dimensional arrays in the discussion. For $n$ dimensions, there is a $\mathrm{PTD}_i$ per dimension. Thus, $A(s(i))$ should be thought of as $A(\ldots, s(i), \ldots)$.

[4] The more general case of affine subscript functions over enclosing loop variables can also be handled. If $i$ is the innermost loop variable that appears in the subscript, then $s(i) = a_1 i + a_0$ still holds by letting $a_0$ be an affine function over the rest of the loop variables. Some additional conditions for loop partitioning must be observed when such subscripts are allowed.

### 3.2 Computation Partitioning

The compiler applies the *owner computes rule* which states that the processor owning a data item performs all computations for that item. An efficient implementation must avoid the run-time overhead of computing ownership. For a loop nest, this is done by reducing the loop bounds according to the Reduced Iteration Set [8] (RIS) of the loop. This is the union of the ACCESS sets of the left-hand side (*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. The RIS is in the global iteration space **GI**. If used directly, an index translation $\tau_p$ is required at every iteration to transform $s(i)$ from the global index space **GN** into the local space **LN** of $p$: DO ($i\ \in$ RIS) A($\tau_p$(s(i)))=$\cdots$ This can be costly if |RIS| is large. A better approach is to obtain LRIS = $\lambda_p$(RIS), which is in the local iteration space **LI** of $p$. The loop becomes DO ($i \in$ LRIS) A(s(i))=$\cdots$ and is more efficient since |RIS| $\tau_p$ operations are saved at run time.

If the ACCESS set of a *lhs* in the loop is a *proper* subset of the RIS, then the corresponding statement must be *masked* [8] 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 coalesced and extracted out of a loop) are performed automatically by the compiler. Details about the conditions and algorithms to compute RIS, LRIS, and masks can be found in [13].

### 3.3 Communication Determination

To find the necessary communication among processors, the compiler computes several *communication sets* for each right-hand side (*rhs*) reference. Details of these sets and the algorithms to calculate them are found in [13, 14]. A brief description follows.

**RECEIVE and SEND Global Iteration Sets** The first communication sets computed are the RECEIVE and SEND sets, both in the global iteration space **GI**. For an assignment statement $S$ with a *lhs* reference $L$ and a *rhs* reference $R$, RECEIVE$(R, p)$ is the set of global iterations for which $p$ owns $L$ but not $R$. Thus, $p$ must *receive* this data via interprocessor communication to execute $S$. Similarly, SEND$(R, p)$ is the set of global iterations for which $p$ owns $R$ but not $L$, and hence must *send* the data to the owner processor of $L$ for it to execute $S$. Thus, we have:

$$\begin{aligned} \textsc{Receive}(R,p) &= \textsc{Access}(L,p) - \textsc{IterImage}(R,p) \\ \textsc{Send}(R,p) &= \textsc{Access}(R,p) - \textsc{IterImage}(L,p) \end{aligned}$$

**IN and OUT Local Index Sets** To generate communication, the array sections to be communicated must be determined. The RECEIVE and SEND sets are translated from the global iteration space **GI** to the local index space **LN** of each processor $p$, obtaining the IN and OUT sets, respectively. First the subscript function $s(i)$ is applied to translate the sets from global iterations **GI** to global indices **GN**, and then the index translation function $\tau_p$ is used to take the sets from global indices **GN** to local indices **LN** of $p$:

$$\begin{aligned} \textsc{In}(R,p) &= \tau_p(s(\textsc{Receive}(R,p))) \\ \textsc{Out}(R,p) &= \tau_p(s(\textsc{Send}(R,p))) \end{aligned}$$

**PARADIGM Code Example** It is also of interest to examine the actual code generated. Figure 6 shows a serial program for a two-dimensional Jacobi iteration. Figure 7 shows the resulting optimized node program supporting a variable number of iPSC processors with a minimum configuration of four nodes. Notice the use of symbolic block sizes (highlighted) whose values are computed at run time.

```
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 6: Serial Version of Jacobi

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

Figure 7: Jacobi for Variable Number of Processors, with a Minimum $2 \times 2$ Configuration

# 4 Conclusions

The purpose of this work is to provide a uniform and efficient way of describing distributed data or computation. The main advantages of the PTD over many other existing representations can be summarized as follows: (1) a single PTD describes all partitions of a region regardless of the shape and size of each partition; (2) a wide range of regions are possible with arbitrary numbers of dimensions, arbitrary boundary angles (not just multiples of 45°), and nonconvex shapes; (3) the PTD supports symbolic set functions and domain transformations, essential for parallelizing compilers for distributed memory; and (4) symbolic block sizes make it possible to compile for a variable number of processors.

**Acknowledgements:** The authors would like to thank the reviewers 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.

# References

[1] S. P. Amarasinghe and M. S. Lam. "Communication Optimization and Code Generation for Distributed Memory Machines," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 126–138, June 1993.

[2] V. Balasundaram. "A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: the Data Access Descriptor," *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

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

[4] D. Callahan and K. Kennedy. "Analysis of interprocedural side effects in a parallel programming environment," in *Proceedings of the First ACM International Conference on Supercomputing*, pages 138–171, Athens, Greece, 1987.

[5] B. Chapman, P. Mehrotra, and H. Zima. "Programming in Vienna Fortran," in *Third Workshop on Compilers for Parallel Computers*, pages 145–164, 1992.

[6] M. Gupta and P. Banerjee. "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

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

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

[9] J. G. Holm, A. Lain, and P. Banerjee. "Compilation of Scientific Programs into Multithreaded and Message Driven Computation," to appear in the *1994 Scalable High Performance Computing Conference*, 1994.

[10] C. Koelbel. "Compile-Time Generation of Regular Communications Patterns," in *Proceedings of Supercomputing '91*, pages 101–110, Albuquerque, NM, November 1991.

[11] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee. "Compiler Optimizations for Distributed Memory Multicomputers used in the PARADIGM Compiler," to appear in the *1994 International Conference on Parallel Processing*, August 1994.

[12] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. "A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," to appear in the *1994 International Conference on Parallel Processing*, 1994.

[13] 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.

[14] E. Su, D. J. Palermo, and P. Banerjee. "Automating Parallelization of Regular Computations for Distributed Memory Multicomputers in the PARADIGM Compiler," in *Proceedings of the 1993 International Conference on Parallel Processing*, pages II:30–38, August 1993.

[15] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. "Integrating scalar optimizations and parallelization," in U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua editors, *Languages and Compilers for Parallel Computing*, pages 137–151, Berlin, Germany, 1992. Springer-Verlag.