Syracuse University

SURFACE

Electrical Engineering and Computer Science - Technical Reports

College of Engineering and Computer Science

5-1995

A Domain-Specific Parallel Programming System II. Automatic Data Partitioning

Elaine Wenderholm
Syracuse University

Follow this and additional works at: https://surface.syr.edu/eecs_techreports

Part of the Computer Sciences Commons

Recommended Citation

Wenderholm, Elaine, "A Domain-Specific Parallel Programming System II. Automatic Data Partitioning" (1995). *Electrical Engineering and Computer Science - Technical Reports*. 141.
https://surface.syr.edu/eecs_techreports/141

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science - Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

SU-CIS-95-2

A Domain-Specific Parallel Programming System

II. Automatic Data Partitioning

Elaine Wenderholm

May 1995

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, NY 13244-4100

A Domain-Specific Parallel Programming System

II. Automatic Data Partitioning

Elaine Wenderholm
School of Computer and Information Science
Syracuse University, NY 13244
wender@top.cis.syr.edu

May 1995

Abstract

$\mathcal{E}m$ is a high-level programming system which puts parallelism within the reach of scientists who are not sophisticated programmers. $\mathcal{E}m$ both restricts and simplifies the programming interface, and thereby eases both the conceptual task of the programmer and the analytical task of the compiler.

The $\mathcal{E}m$ compiler performs automatic data structure definition, scheduling and data partitioning.

This document presents the automatic data partitioning algorithm used in $\mathcal{E}m$.

1 Introduction

$\mathcal{E}m$ is a high-level programming system which puts parallelism into the hands of scientists who are not sophisticated programmers. $\mathcal{E}m$ both restricts and simplifies the programming interface, and thereby eases both the conceptual task of the programmer and the analytical task of the compiler.

There are several examples of successful specialized programming systems, two of which are financial spreadsheets such as Lotus, and symbolic computation systems such as Mathematica. Each of these tools has allowed a community of users to write applications that previously required specialist programmers. Many users simply would be unable to develop such applications without the use of these specialized software systems. These tools share three characteristics:

1. Each addresses a restricted and well-defined problem domain.
2. The interface to each tool is designed to be intuitive to the target community.
3. Features from declarative and functional programming are incorporated into the language, thereby freeing the user from programming details, and the need to manage storage and other machine resources.

The design of $\mathcal{E}m$ exhibits these same characteristics:

1. $\mathcal{E}m$'s problem domain centers on the class of simulation problems which is statically decomposed, has communication localized to a fixed neighborhood, and is loosely synchronous, i.e., time is incremented synchronously after all spatial components are updated.

2. $\mathcal{E}m$ provides a high-level interface with a domain-specific library. The library can be customized to a specific area of scientific investigation.

3. $\mathcal{E}m$ programs are almost purely functional. This relieves the programmer of the need to manage storage and other machine resources, a most difficult task when writing parallel programs.

This paper proceeds as follows. Section 2 defines the semantics of the loop nest in an $\mathcal{E}m$ program. Section 3 describes the types of problems which may be solved using $\mathcal{E}m$. In Section 4, the main features of the data partitioning strategy are presented, along with definitions necessary for understanding the rest of the paper. A three-dimensional iteration space is used, in Section 5, as an example to present the algorithm to minimize communication. The algorithm and proof are presented in full generality in the Appendix. Calculation of communication weights is presented in Section 6. Section 7 presents examples of the partitioning algorithm, and is followed by concluding remarks.

2 Loop Structure

An $\mathcal{E}m$ program contains one loop. Figure 1 shows an $\mathcal{E}m$ loop and its semantically equivalent loop nest. The outermost loop (time) is sequential, and enforces synchronization at the end of each iteration. The set of n-1 inner loops have no loop-dependent dependences and may be written as *DoAll* loops. These inner loops are parallel and generally update large data sets. The data updates are performed relative to a statically defined neighborhood. The code body, S, resides within the innermost loop. It consists of procedure calls and may contain conditional statements.

Associated with each $\mathcal{E}m$ procedure is a *procedure summary.* The procedure summary contains data access information as read, write, +reduce (sum reduction) and ∗reduce

𝓔m Loop

```
...
loop
    ...
    S
    ...
end loop
```

Semantic Equivalent

```
L_1: Do time = ltime, utime
L_2:     DoAll I_2 = $L_2$, $U_2$
             ...
L_n:         DoAll I_n = $L_n$, $U_n$
                 ...
                 S
                 ...
             End DoAll I_n
             ...
         End DoAll I_2
     End Do time
```

Figure 1: 𝓔*m* Loop and Semantically Equivalent Loop Nest

(product reduction). Each data access is specified as relative address (offset) to the current cell, or as an absolute address of the iteration space. The set of array accesses for each cell, which contains one or more array variables, is referred to as the cell's *stencil* [RAP87, FJL$^+$88, HA90]. Since a data access is a component-wise definition of communication, the stencil may be used to quantify communication by component.

3 Problem Domain

𝓔*m* is a system which is used to solve problems in which array accesses are localized to a statically defined neighborhood.

The iterative solution of simple elliptical partial-differential equations provides an easy example of a nearest neighbor problem. Given Laplace's equation

$$\frac{\partial^2 \phi}{\partial x^2}(x,y) + \frac{\partial^2 \phi}{\partial y^2}(x,y) = 0$$

the central-difference equation is

$$4\phi_{(i,j)} - \phi_{(i+1,j)} - \phi_{(i-1,j)} - \phi_{(i,j+1)} - \phi_{(i,j-1)} = 0$$

where i, j are indices over the grid. Using the Jacobi iteration, the approximation at iteration k of a grid point at (i, j) is the average of the neighboring values at iteration k-1.

$$\phi(i,j)^k = .25 * (\phi(i+1,j)^{k-1} + \phi(i-1,j)^{k-1} + \phi(i,j+1)^{k-1} + \phi(i,j-1)^{k-1})$$

These array accesses have center symmetry, and an optimal partitioning of a square data space is square [FJL$^+$88].

The partitioning problem is not always so obvious: array accesses need not be center symmetric, e.g., when forward-difference and backward-difference methods are used; or the iteration space may not be regular.

4 Data Partitioning

This paper presents a new data partitioning strategy. In particular,

1. There is no restriction to a square iteration space as in [HA90, RAP87]; it is generalized to a non-regular n-dimensional iteration space $D_1 \times D_2 \times \ldots \times D_n$. In practical physical simulations, however, n does not exceed four.

2. One or more data sets may be used.

3. Conditional procedure invocation, whose execution count may be determined at compile time, is incorporated into the strategy.

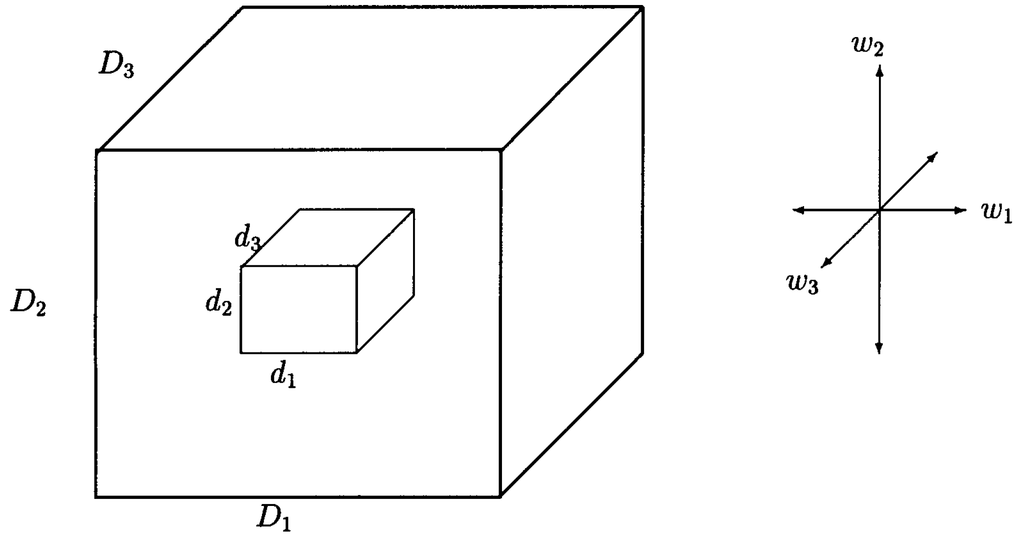

Figure 2: Iteration Space $D_1D_2D_3$, Block $d_1d_2d_3$, and Communication Weights w_1, w_2, w_3

4. Data sets which have an iteration space of less than n dimensions are accommodated. This is motivated by practical ecological modelling problems. Ecological modelling generally requires several state variables, and often requires state variables which span different dimensions. As a case in point, the wetland example presented in [WB94] defines a 3-dimensional iteration space: `water` spans the entire space; `ducks` span only the `x-y` plane at `z=1`.

The partitioning strategy assumes a distributed multiprocessor system. The number of processors is a power of two. The data is partitioned statically across processors. The loop nest is the type shown in Figure 1.

4.1 Definitions

Definition 1 *Given a loop $L = L_1, \ldots, L_n$, and a statement S within L, an* iteration vector $\vec{i} = (i_1, \ldots, i_n) \in \mathbf{Z}^n$, $i_j \in [L_j : U_j]_{(1 \le j \le n)}$ *corresponds to an execution instance of S [ZC91].*

Definition 2 *The* iteration space, *It^n, of loop $L = L_1, \ldots, L_n$ is a finite region in n-dimensional discrete Cartesian space whose points correspond one-to-one to iteration vectors [ZC91].*

It follows that: the axes of the iteration space lexicographically correspond one-to-one to the loop nest; all edges incident on a vertex are mutually orthogonal.

Definition 3 *$D_{j(1 \le j \le n)}$ denotes the* span, *$U_j - L_j + 1$, of the j^{th}-dimension of It^n.*

Definition 4 *By P, denote the number of processors. A* partition *of the iteration space It^n is a set, $block^b$ $_{(1 \le b \le P)}$, of discrete Cartesian subspaces such that*

1. *the subspaces are mutually disjoint, $block^{b_1} \cap block^{b_2} = \emptyset$, for $(1 \le b_1 < b_2 \le P)$,*
2. *exhaustive, $\bigcup_{1 \le b \le P} block^b = It^n$,*
3. *and nonempty.*

Definition 5 *A* block^b $_{(1 \le b \le P)}$ *consists of a set of iteration vectors $\vec{i^b} = (i_1^b, \ldots, i_n^b) \in \mathbf{Z}^n$, $i_j^b \in [l_j^b : u_j^b]$, $L_j \le l_j^b \le u_j^b \le U_j$, $_{(1 \le j \le n)}$.*

Definition 6 *$d^b_{j(1 \le j \le n)}$ denotes the* span, *$d_j^i = u_j^b - l_j^b + 1$ of the j^{th}-dimension of $block^b{}_{(1 \le b \le P)}$.*

Definition 7 *A* face, $face_i^b$, perpendicular to the i^{th} component *of* $block^b$, $(1 \le i \le n)$, *is that subset of* $block^b$, $\{\vec{i^b} = (i_1^b, \ldots, f_i^b, \ldots, i_n^b) : i_j^b \in [l_j^b : u_j^b]$ $(1 \le j \ne i \le n)$ *and either* $f_i^b = l_i^b$ *or* $f_i^b = u_i^b\}$. *The face in which* $f_i^b = l_i^b$ *is the* lower face, *denoted* $lface_i^b$; *the face in which* $f_i^b = u_i^b$ *is the* upper face, *denoted* $uface_i^b$. *A face has dimension* n-1.

The faces are the extremal (n-1)-dimensional subspaces bounding the n-dimensional block. There are two parallel faces per dimension, which are orthogonal to all other faces.

The number of extremal subspaces of dimension $k < n$ is determined by first choosing k dimensions in the space. There are $\binom{n}{k}$ ways to chose k dimensions. Fixing the remaining $n - k$ dimensions to one of the extremal values, gives the number of $k < n$ subspaces as $\binom{n}{k} 2^{n-k}$. For example the number of faces in 3-dimensional space is $2n = 6$. In general, since there are two parallel faces per dimension, this results in a total of $2n$ faces for an n-dimensional block.

Definition 8 *Given vectors* $\vec{p}_1$, $\vec{p}_2$ *in* It^n, $\vec{p}_o$ *is an* offset vector *of* $\vec{p}_1$ *if* $\vec{p}_1 + \vec{p}_o = \vec{p}_2$.

Definition 9 *An* access vector, $\vec{av} = [av_1, \ldots, av_n]$, *is an ordered tuple where each component,* av_i, $(1 \le i \le n)$, *of* $\vec{av}$ *is either*

1. *an offset address,* $av_i \in \mathbf{Z}$,

2. *or an absolute address* $av_i \in \{lb[+E], ub[-E]\}$, $E \in \mathbf{Z}$, *and* $L_i \le av_i \le U_i$.

Let v be a variable, and let p be a procedure invocation in an $\mathcal{E}m$ program. By $av(v)$, denote an *access vector for variable* v. By $av(v, p)$, denote an *access vector for variable* v *in procedure invocation* p. By $av_i(v)$, denote the *ith component of* $av(v)$. By $av_i(v, p)$, denote the *ith component of* $av(v, p)$. A stencil will be regarded as a set of access vectors.

Definition 10 *The* communication weight *is a vector* $\vec{w} = (w_1, \ldots, w_n) \in \mathbf{N}^n$. [1]

Definition 11 *The* communication weight along component i, w_i, *is the weight assigned to* $face_i$. [2]

Figure 2 shows a 3-dimensional iteration space of $D_1 \times D_2 \times D_3$. Within this space is block $d_1 d_2 d_3$. Communication may occur at block boundaries across the upper and lower faces of $d_1 d_2$, $d_2 d_3$, and $d_3 d_1$. The vectors w_1 w_2 and w_3 denote the dimension along which communication may occur. Section 6 presents the computation of communication weights.

Definition 12 *The* surface area of face $face_i$ *is* $\mathcal{S}_{f_i} = \prod\limits_{\substack{j \neq i \\ 1 \leq j \leq n}} d_j$.

Definition 13 *The* weighted surface area of face $face_i$ *is* $w_i \mathcal{S}_{f_i} = w_i (\prod\limits_{\substack{j \neq i \\ 1 \leq j \leq n}} d_j)$.

It can be seen that positive-valued w_i cross $uface_i$, and negative-valued w_i cross $lface_i$.

5 Minimizing Communication

The objective in any data partitioning strategy is to minimize the ratio of communication to computation. Additionally, in order to have the workload balanced between all processors, each processor should perform an identical amount of computation. In general, communication is minimized by minimizing the "surface area" of the data

[1] Definition 19 will refine this to $\vec{w} \in (\frac{\mathbf{N}}{\mathbf{N}^{+}})^n$.

[2] The superscript denoting the block, as in $block^b$, d^b_j and $face^b_j$, will be omitted whenever no confusion exists.

space; load balancing is accomplished by partitioning the data space into equal "volumes" across processors. [3]

This partitioning strategy uses the weighted surface area. Using Figure 2 as the example, and assuming all communication weights are nonzero, the weighted surface area of each block is the sum of the weighted surface areas of all faces:

$$\mathcal{S}_b = 2w_1 d_2 d_3 + 2w_2 d_1 d_3 + 2w_3 d_1 d_2 \tag{1}$$

The data is to be partitioned equally among the processors, so that the volume $\mathcal{V}$ of each block is

$$d_1 d_2 d_3 = \mathcal{V} = \frac{D_1 D_2 D_3}{P}. \tag{2}$$

The problem is to minimize the surface area per block

$$\frac{\partial \mathcal{S}_b}{\partial d_1} dd_1 + \frac{\partial \mathcal{S}_b}{\partial d_2} dd_2 + \frac{\partial \mathcal{S}_b}{\partial d_3} dd_3 = 0 \tag{3}$$

subject to the constraint imposed by equation 2, i.e.,

$$\varphi = d_1 d_2 d_3 - \mathcal{V} = 0. \tag{4}$$

Using the technique of Lagrange multipliers [SR58], equations 3 and 4 yield

$$\begin{array}{rclclclcll}
\frac{\partial \mathcal{S}_b}{\partial d_1} + \lambda(\frac{\partial \varphi}{\partial d_1}) & = & & + & w_2 d_3 & + & w_3 d_2 & + & \lambda(d_2 d_3) & = 0 \\
\frac{\partial \mathcal{S}_b}{\partial d_2} + \lambda(\frac{\partial \varphi}{\partial d_2}) & = & w_1 d_3 & + & & + & w_3 d_1 & + & \lambda(d_1 d_3) & = 0 \\
\frac{\partial \mathcal{S}_b}{\partial d_3} + \lambda(\frac{\partial \varphi}{\partial d_3}) & = & w_1 d_2 & + & w_2 d_1 & + & & + & \lambda(d_1 d_2) & = 0
\end{array}$$

To simplify these equations, multiply the first by d_1, the second by d_2, and the third by d_3. The result is

$$\begin{array}{rcrcrcll}
 & & w_2 d_1 d_3 & + & w_3 d_1 d_2 & + & \lambda(d_1 d_2 d_3) & = 0 \\
w_1 d_2 d_3 & + & & + & w_3 d_1 d_2 & + & \lambda(d_1 d_2 d_3) & = 0 \\
w_1 d_2 d_3 & + & w_2 d_1 d_3 & + & & + & \lambda(d_1 d_2 d_3) & = 0
\end{array}$$

[3] It should be noted that blocks are always of dimension n. In discussions of partitioning strategies, one commonly encounters phrases such as "...in an n by n domain, a one-dimensional decomposition is ...and the two-dimensional case is...". This dimension refers to the number of dimensions over which communication occurs. In the case where there are $k < n$ dimensions of communication, the iteration space itself forms $n - k$ boundaries, obviating the need for communication. A toroidal It^n has no boundary conditions, and communication is of dimension n.

By subtracting the equations "round robin", i.e., first from second, second from third, and third from first, the equations are simplified to the ratios

$$\frac{d_1}{d_2} = \frac{w_1}{w_2}; \quad \frac{d_2}{d_3} = \frac{w_2}{w_3}; \quad \frac{d_3}{d_1} = \frac{w_3}{w_1}. \tag{5}$$

Solving for d_1 using equations 5 and 2,

$$d_1 = d_2\frac{w_1}{w_2} = \frac{\mathcal{V}w_1}{d_1d_3w_2} = \frac{\mathcal{V}(w_1)^2}{(d_1)^2w_2w_3},$$

and

$$(d_1)^3 = \frac{\mathcal{V}(w_1)^2}{w_2w_3}.$$

Substituting for $\mathcal{V}$,

$$(d_1)^3 = \frac{D_1D_2D_3(w_1)^2}{w_2w_3}.$$

Defined logarithmically (the base is irrelevant),

$$\log d_1 = \frac{1}{3}\log\frac{(w_1)^2D_1D_2D_3}{Pw_2w_3}$$

and similarly for d_2 and d_3,

$$\log d_2 = \frac{1}{3}\log\frac{(w_2)^2D_1D_2D_3}{Pw_1w_3}$$

$$\log d_3 = \frac{1}{3}\log\frac{(w_3)^2D_1D_2D_3}{Pw_1w_2}.$$

so that

$$d_1 = \text{antilog}(\frac{1}{3}\log\frac{(w_1)^2D_1D_2D_3}{Pw_2w_3})$$

and similarly for d_2 and d_3.

A proof is provided in the appendix for a general n-dimensional iteration space, where communication weights may be zero. In [HA90], zero communication weights are not considered.

The total number of processors, P, is an integer power of 2, and $P = p_1 \cdot \ldots \cdot p_n$, where p_i is the number of subdivision of component i, $(1 \leq i \leq n)$. Therefore each p_i is also an integer power of 2. The p_i are calculated using $p_i = \frac{D_i}{d_i}$.

In those cases where the calculated p_i is not an integer power of 2, the p_i are approximated by rounding up/down to the nearest integer power of 2, and calculating a non-minimal surface area. The chosen values of p_i are those which give the smallest calculated surface area. In the worst case, n dimensions must be estimated, resulting in an upper bound of 2^n comparisons of calculated surface areas. In most cases, n is quite small.

6 Quantification of Communication Weight

The communication weight is computed using a *min-max construction* [HA90], and is considered a "best case" estimate since it assumes that all data external to a block need be communicated only once to satisfy all internal references. This construction is suitable to an $\mathcal{E}m$ program. $\mathcal{E}m$ programs are highly functional, and $\mathcal{E}m$ array access rules require that all updates are local to the cell.

6.1 Simplest Programming Model

The simplest programming model in $\mathcal{E}m$ is comprised of one array variable spanning the iteration space, and one procedure inside the loop. In addition, there is no conditional procedure execution. This insures identical communication between blocks. Communication weight calculation for this model is identical to [HA90]:

Definition 14 *The* simplest communication weight along component *i, w_i, is defined*

$$\begin{aligned} w_i &= \max(\{av_i : av_i \geq 0\} \cup \{0\}) + \\ &\quad |\min(\{av_i : av_i < 0\} \cup \{0\})| \end{aligned}$$

where av_i denotes an access vector along component i, and $(1 \leq i \leq n)$

6.2 $\mathcal{E}m$ Program Model

An $\mathcal{E}m$ program consists of one or more procedure invocations within a loop, and one or more array variables.

For this model, communication weight is extended to be a function of variable. Let Var be the set of array variables.

Definition 15 *The* variable communication weight, *$w_i(v)$, along component i for variable $v \in V$ is*

$$\begin{aligned} w_i(v) &= \max(\{av_i(v) : av_i(v) \geq 0\} \cup \{0\}) + \\ &\quad |\min(\{av_i(v) : av_i(v) < 0\} \cup \{0\})| \end{aligned}$$

where $(1 \leq i \leq n)$.

Obviously, the component-wise communication weight incurred for several variables is additive.

Definition 16 *The* communication weight w_i for component i over variables $v \in V$ *is*

$$w_i = \sum_{v \in V} w_i(v)$$

Definition 17 *Since expansion of the scalar w_i into vector $|\vec{w}| = n$ is $\vec{w}_i = (0_1, \ldots, w_i, \ldots, 0_n)$, $(1 \leq i \leq n)$, the* overall communication weight, *$\vec{w}$ is*

$$\vec{w} = \sum_{1 \leq i \leq n} \vec{w}_i$$

6.3 Iteration Subspaces

An $\mathcal{E}m$ procedure summary may contain array accesses which define a subspace of the iteration space.

Definition 18 *A* slice *is a subset of the iteration space over which an array variable is accessed.*

Procedures may be required which only handle iteration space boundary conditions. Therefore, array accesses performed by such procedures occur only at boundaries of the iteration space. Section 4 provides a good example: given an iteration space `(x,y,z)`, variable `v` spans the `x-y` plane only at `z=1`. An access vector for `v` is, say, $av(v) = [0, 2, lb]$, where lb denotes the lower bound of the third component of the iteration space.

Assuming a span, D_3, for the third component, this array access may be weighted as $(1/D_3)[0, 2, 0] = [0, 2/D_3, 0]$. In other words, a slice has width 1 and contributes $1/D_3$ to the communication of the access vector, and the remaining $D_3 - (1/D_3)$ "slices" have no communication.

Conversely, consider an access vector for variable v, $av(v)$, where $av_i(v)$ is an offset address. Then $av_i(v)$ applies to the span of component i; there are D_i "slices", each contributing $1/D_i$ to the communication. The sliced communication weight is simply $(D_i)(1/D_i) \times av(v) = av(v)$.

Definition 19 *The function* slice *is defined:*

$$slice : (v \times i) \times av \mapsto \{1, 1/D_i\},$$

and

$$slice(v,i)[\ldots, av_i, \ldots] = \begin{cases} 1/D_i & \text{if } av_i = lb[+E] \text{ or } av_i = ub[-E] \\ 1 & \text{otherwise} \end{cases}$$

where, av is an access vector $[av_1, \ldots, av_n]$ *for variable* $v \in V$*, and* $(1 \leq i \leq n)$.

Lastly, a sliced access vector must be transformed into a nonsliced vector.

Definition 20 *The function '*' is defined:*

$$* : av \mapsto av$$

and

$$*av = \begin{cases} 0 & \text{if } av_i = lb[+E] \text{ or } av_i = ub[-E] \\ av_i & otherwise \end{cases}$$

where av is an access vector, and $1 \leq i \leq n$.

It is then possible to weight the sliced access vectors and use weighted access vectors to calculate the communication weights.

$$wav_i(v) \quad = \quad (slice(v,i)av(v))(*av(v))$$

where $(1 \leq i \leq n)$. This definition states the weighted slice of an access vector with one "slice" at i. However in general, an access vector may be sliced over more than one component. Obviously, the *sliced area* of the access vector is simply calculated as the product of the slices over all components.

Definition 21 *Let*

$$prod(av_i(v)) = (\prod_{1 \leq i \leq n} slice(v,i)av(v))(*av(v)).$$

Then the weighted access vector, $wav_i(v)$, *over component i for variable* $v \in V$ *is*

$$\begin{aligned} wav_i(v) \quad = \quad & \max(\{prod(av_i(v) : prod(av_i(v)) \geq 0\} \cup \{0\}) + \\ & |\min(\{prod(av_i(v) : prod(av_i(v)) < 0\} \cup \{0\})| \end{aligned}$$

Now, Definition 16, which defines $w_i(v)$ as a function of access vectors, may be defined as a function of weighted access vectors, viz.,

$$\begin{aligned} w_i(v) &= \max(\{wav_i(v) : wav_i(v) \geq 0\} \cup \{0\}) + \\ & \quad |\min(\{wav_i(v) : wav_i(v) < 0\} \cup \{0\})| \end{aligned}$$

The option exists in the $\mathcal{E}m$ compiler to select the degree of accuracy for determining the communication weight by either incorporating or not incorporating sliced access vectors into the calculation.

7 Examples

This section presents examples of three kinds. First, array accesses where all communication weights are greater than zero are discussed. [4] In fact, only those problems whose weights are not zero have been considered by the researchers previously cited.

Next, communication weights of zero are discussed. And lastly, a simple example is presented of iteration spaces slices, and how slices affect the data partition.

7.1 Non-Zero Weighted Communication

Frequently used discretization stencils [RAP87] are shown in Figure 3, along with their communication weight vectors determined by the algorithm herein. The set of access vectors for a 7-point star stencil is

$$\{[-1,1],[0,1],[1,0],[1,-1],[0,-1],[-1,0]\}$$

and the w_i are calculated

$$w_1 = \max(\{-1,0,1,1,0,-1\} \cup \{0\}) + |\min(\{-1,0,1,1,0,-1\} \cup \{0\})| = 2$$

[4] By definition, communication weights are not negative valued.

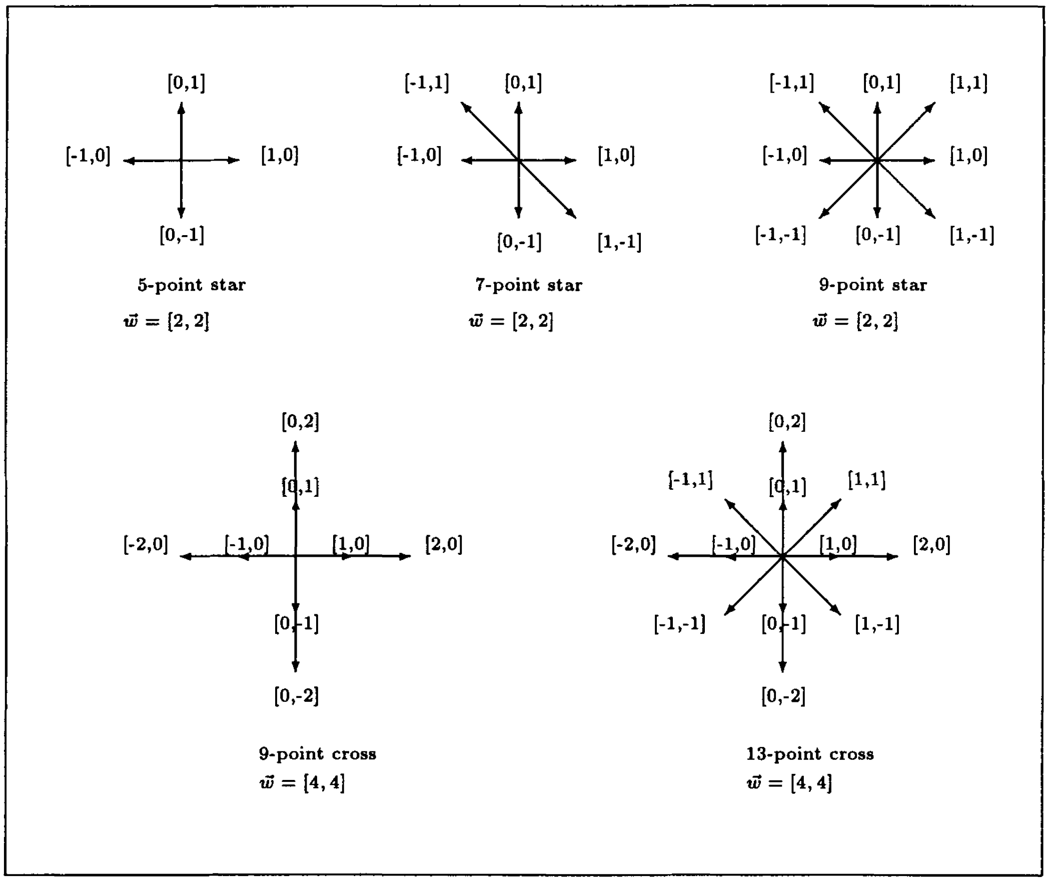

Figure 3: Discretization Stencils and Communication Weight Vectors

$$\begin{aligned} w_2 &= \max(\{1,1,0,-1,-1,0\} \cup \{0\}) + |min(\{1,1,0,-1,-1,0\} \cup \{0\})| = 2 \\ \text{hence, } \vec{w} &= [2,2] \end{aligned}$$

The communication wieghts of these stencils share a common property: all components are in a 1 : 1 ratio, indicating a center symmetric communication pattern. It is possible to take exception to this statement by noting that, e.g., a 7-point star stencil is not center symmetric due to the diagonal offsets which contain both horizontal and vertical components. However, as proven in [HA90], this additional communication

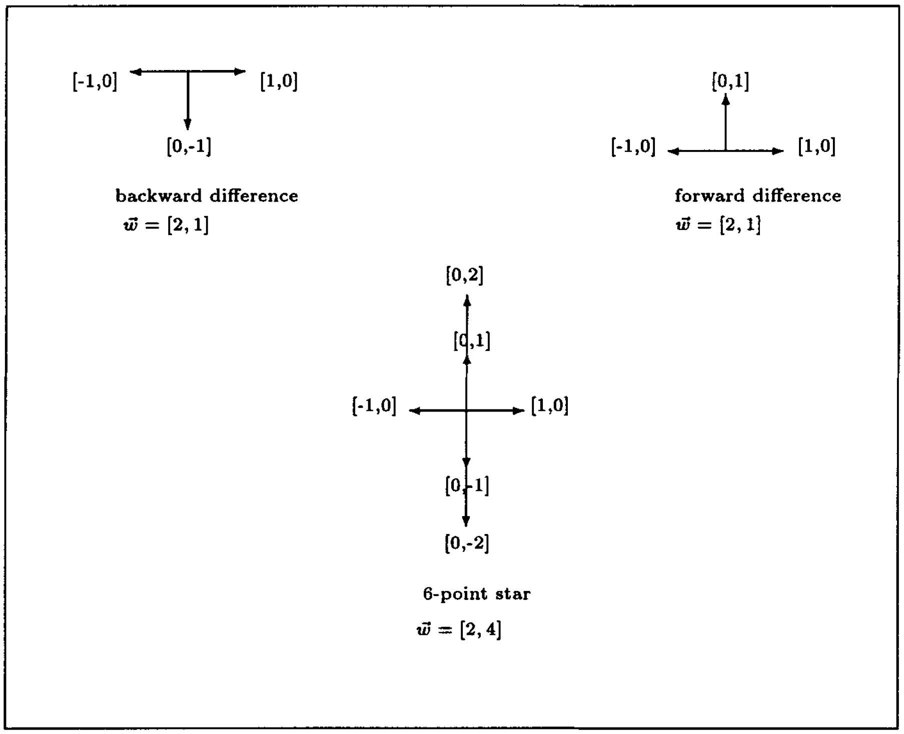

Figure 4: Nonuniform Discretization Stencils and Communication Weight Vectors

results in a constant error term, is independent of the dimension of the rectangular partition, and whose magnitude is independent of optimization technique.

A naive approach to communication would construct four messages for the north, south, east, and west *faces* of the block, and two additional messages for the two remaining diagonal *elements* of the block. But as out in [FJL$^+$88][5], separate communication for diagonal elements is not necessary for Cartesian partitions: a maximum of $2n$ messages are required. All stencils in Figure 3 require four messages.

Non-center symmetric stencils are shown in Figure 4. The communication weight vector for the forward-difference and backward-distance stencil is $\vec{w} = [2, 1]$; the communication weight vector for the 6-point star stencil is $\vec{w} = [2, 4]$. An optimal

[5] Ch. 16

	nproc = 16			nproc = 32		
defined $\vec{w}$	partition	result $\vec{w}$	%error	partition	result $\vec{w}$	%error
[1,0,1]	[4,1,4]	[1,0,1]	0	[8,1,4] ([4,1,8])	[1,0,1]	5.6
[1,1,1]	[2,4,2]	[1,1,1]	5.8	[4,2,4]	[1,1,1]	5.0
[1,2,1]	[4,1,4]	[1,2,1]	5.0	[4,2,4]	[1,1,1]	0
[1,3,1]	[4,1,4]	[1,3,1]	0.9	[4,2,4]	[1,3,1]	1.9
[1,4,1]	[4,1,4]	[1,4,1]	0	[4,2,4]	[1,4,1]	5.8
[1,5,1]	[4,1,4]	[1,0,1]	0	[4,2,4]	[1,5,1]	4.4
[1,6,1]	[4,1,4]	[1,0,1]	0	[8,1,4] ([4,1,8])	[1,0,1]	5.6

Table 1: Comparison of w_2 for $It^3 = n \times n \times n$

partition [HA90] is achieved when the ratio of the block faces are 2 : 1, 1 : 2 resp.

It should be noted that communication vectors, say, $[2, 1]$ and $[4, 2]$, have identical ratios, and produce identical partitions. Their difference reflects the *quantity* of data to be communicated across each face. This is a separate topic and is outside the scope of this paper.

7.2 Zero-Weighted Communication

Appendix A presents the proof for communication weights which include zero-valued weights. Lemma A proves there are two situations in which a communication weight, w_i, is zero:

1. $w_i = 0$ by definition.
2. The minimization determines that $d_i = D_i$, so that no communication occurs along component i, and results in a computed communication weight $w_i = 0$.

Table 7.1 demonstrates this Lemma. The first row shows a weight whose second component is zero by definition. In the resulting partition, the second component is 1, showing that the processor boundary is the iteration space boundary. The following rows demonstrate the effect of increasing the weight along component two: there is a point at which the magnitude of the communication along component becomes so large that the algorithm reduces the number of dimensions for communication from three to two.

7.3 Iteration Subspaces

As discussed in Section 6.3, the unique aspect of $\mathcal{E}m$ is the ability to determine communication weights for subspaces of the iteration space.

Consider a 3-dimensional iteration space whose components are denoted, respectively, as (x, y, z). Next, consider a procedure, $P(a, b)$, which executes over all of x and y at $z = 1$. The code for the correct execution of this procedure requires the conditional execution of P. The conditional execution of P is an example of *control dependence.*

This code may be written in several ways. For example, a programmer may write the code with the loop nest either enclosed within the conditional statement, or enclosing the conditional statement. This is shown in Figure 5.

An optimizing compiler may or may not be able to detect control dependence, depending on the way in which the code is written. This is especially true for interprocedural analysis.

The $\mathcal{E}m$ language overcomes these difficulties:

1. $\mathcal{E}m$ enforces a standardized calling sequence for procedures.
2. The procedure summary of $\mathcal{E}m$ encodes control dependence as array access vectors.

Specifically, any access vector component which is not an offset address specifies a control dependence. The combination of the procedure summary and standard

```
do x = x1, xn
   do y = y1, yn
      do z = z1, zn
         if (z .eq. 1) then
            call P(a,b)
         endif
      enddo
   enddo
enddo
```

```
do x = x1, xn
   do y = y1, yn
      do z = z1, zn
         call P(a,b)
      enddo
   enddo
enddo
...
and,
...
Subroutine P(a,b)
...
   if (z .eq. 1) then
      ...
   endif
```

Figure 5: Coding Styles for the Conditional Execution of Procedure $P(a, b)$

	water	*soil_structure*
$\vec{w}$	$[2, 2, 10]$	$[4, 8, 0]$
$\vec{wsl}$	$[2, 2, 10]$	$(0.5)[4, 8, 0]$

Table 2: $\vec{w}$ and $\vec{wsl}$ for *water* and *soil_structure*

$It^3 = n \times n \times n, nproc = 64$			
	Case 1	**Case 2**	**Case 3**
$\vec{w}$	$[2, 2, 10]$	$[6, 10, 10]$	$[4, 6, 10]$
partition	$[8, 8, 1]$	$[4, 4, 4]$	$[8, 4, 2]$

Table 3: Partitions for Three Cases in Conditional Array Assignment

procedure interface makes the job for the $\mathcal{E}m$ compiler very easy. It is so easy that $\mathcal{E}m$ generates the code for the loop nest and conditional execution of procedures.

7.3.1 Conditional Execution: An Example

The following example illustrates the effect of conditional array assignment on the calculated data partition.

Consider the variables, *soil_structure*, and *water*, which span a cubic iteration space. The data structures for *soil_structure* and *water* span the entire space. Communication for *water* and *soil_structure* occur across all $face_i$ of the space. However, the communication for *soil_structure* across $face_3$ is required over only one-half the space, and this communication is uniformly distributed.

Table 7.3 displays the result of calculating the unsliced, w, and sliced, wsl, communication weights for these variables from their sets of access vectors. The choice for any compiler is whether or not to include the conditional execution of an array assignment. To demonstrate the effect, three cases are tested:

Case 1: A conditional array assignment is ignored.

The communication weight only for *water* is considered.

$$\vec{w} \;=\; \vec{w}(water) = [2,2,10]$$

Case 2: A conditional array assignment is treated as an unconditional assignment.

The communication weight is calculated using both variables as the sum of the $\vec{w}$ of variables $soil\_structure$ and $water$.

$$\begin{aligned}\vec{w} &= \vec{w}(soil\_structure) + \vec{w}(water)\\ &= [4,8,0] + [2,2,10]\\ &= [6,10,10]\end{aligned}$$

Case 3: A conditional array assignment is treated as a a conditional assignment; sliced communication weights are used.

The sliced communication weight is the sum of the sliced communication weights of variables $water$ and $soil\_structure$. Note that $\vec{w}(water) = \vec{wsl}(water)$.

$$\begin{aligned}\vec{wsl} &= \vec{wsl}(soil\_structure) + \vec{wsl}(water)\\ &= [2,2,10] + (0.5)[4,8,0]\\ &= [4,6,10]\end{aligned}$$

The results are listed in Table 7.3. This example assumed a uniform distribution of communication across $face_3$. However, this need not be the case: array accesses may not be uniformly distributed across the subspace. In this situation there may be an imbalance of communication: the subspace requiring communication may be, say, defined as the "upper half" of $face_3$. Then it would be reasonable to assume a worst-case communication strategy and calculate the data partition using the assumption of Case 2.

8 Conclusion

The data partitioning algorithm is implemented in Prolog [Wie94] as part of the $\mathcal{E}m$ compiler. It may also be invoked separately, and interactively, to provide only data partitioning results. Run separately, the user need specify only three data:

1. iteration space bounds
2. number of processors
3. communication weight vector

The $\mathcal{E}m$ compiler not only performs automatic data partitioning, but also automatically deduces array bounds, which are used to write array declarations. Should the $\mathcal{E}m$ program be changed by the programmer to, say, incorporate an additional (or merely modified) set of procedures which access different portions of array variables, or new array variables, the compiler automatically adjusts the array declarations and data partition. This eliminates the need for the programmer to modify, often erroneously, the program.

The $\mathcal{E}m$ compiler not only uses the access vectors containing slices to compute the sliced communication weight, but also to write the source code for the conditional execution of the procedure. This eliminates the need for the programmer to write conditional statements for bounds checking. An example of the source code generated may be found in [WB94].

The general partitioning algorithm, presented in the Appendix, naturally accommodates arrays of dimension $< n$. As stated above, the $\mathcal{E}m$ compiler determines the array declaration, and hence the array dimensions. As shown in the Appendix, a communication weight $w_i = 0$ implies no communication across $face_i$, which is semantically identical to an array of dimension $k < n$.

The most general programming model supports multiple programs executing on different processors with different data sets. In this case, both the programs and/or the data are partitioned across processors, and is difficult to optimize. This model is beyond the scope of $\mathcal{E}m$ which is designed to support the SPMD (Single Program Multiple Data) Model. However, the procedure summary information specifies array access as a function of variable and procedure name. It is a simple compiler task to map the procedure summary data into communication weight defined as a function of variable name, array access, and procedure invocation. Weights defined as such may be used as input to algorithms which partition not only data, but also procedures, across multiple processors. In this case, the problems solved would still remain restricted to access data within a statically-defined local neighborhood.

References

[FJL$^+$88] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors, Volume I.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[HA90] David E. Hudak and Santosh G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 187–200, Amsterdam, The Netherlands, June 11–15, 1990. Published as ACM SIGARCH Computer Architecture News 18(3).

[RAP87] Daniel A. Reed, Loyce M. Adams, and Merrell L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[SR58] I.S. Sokolnikoff and R.M. Redheffer. *Mathematics of Physics and Modern Engineering.* McGraw-Hill, New York, NY, 1958.

[WB94] Elaine Wenderholm and Micah Beck. A domain-specific parallel programming system. I. Design and application to ecological modelling. Technical Report SCCS-640, Syracuse Center for Computational Science, Syracuse University, September 1994.

[Wie94] Jan Wielemaker. *SWI-Prolog, Version 1.9.0.* University of Amsterdam, The Netherlands, June 1994. (Quintus compatible).

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, NY, 1991.

A Appendix

Lemma A

The weighted surface area of $face_i = 0$ only if $w_i = 0$.

Proof

By definition of block, $\neg\exists_{1\leq j\leq n} d_j = 0$. Therefore it must be that $w_i = 0$. There are two cases in which $w_i = 0$:

case 1 $w_i = 0$ is defined by the stencil.

case 2 The minimization determines that $d_i = D_i$. Then component i of the iteration space is not decomposed. The processor boundary on the i-th component is the iteration space boundary. Thus, no communication occurs along component i, which results in a computed communication weight $w_i = 0$.

□

Problem

Given a n-dimensional space $D_1 \times D_2 \times \ldots \times D_n$ partitioned into P blocks of equal volume $\mathcal{V} = (D_1 \cdot \ldots \cdot D_n) \div P = d_1 \cdot \ldots \cdot d_n > 1$, and communications weights $w_1, w_2, \ldots, w_n$, minimize the weighted surface area per block

$$\begin{aligned}\mathcal{S}_b = {} & w_1 d_2 d_3 \cdot \ldots \cdot d_n + \\ & w_2 d_1 d_3 \cdot \ldots \cdot d_n + \ldots + w_{n-1} d_1 d_2 \cdot \ldots \cdot d_{n-2} d_n + w_n d_1 d_2 \cdot \ldots \cdot d_{n-2} d_{n-1}\end{aligned}$$

constrained by $\varphi = (\prod_{1\leq j\leq n} d_j) - \mathcal{V} = 0$.

Claim 1

Minimization of the weighted surface area per block is achieved by satisfying ratio equations

$$\frac{w_{I(l+1)}}{w_{I(l)}} = \frac{d_{I(l+1)}}{d_{I(l)}} \ (0 \leq l < k) \tag{1}$$

where,

$I = \{i : w_i \neq 0 \text{ and } {\scriptstyle (1 \leq i \leq n)}\}$ is the *ordered index set of weights* w_i, $I(i)$ denotes the $(i+1)$st element of I, k is the length of I, and $(l+1)$ is evaluated mod k.

Proof

$\mathcal{S}_b$ may be written as a sum of products.

$$\mathcal{S}_b = \sum_{1 \leq i \leq n} (w_i)(\prod_{\substack{j \neq i \\ 1 \leq j \leq n}} d_j),$$

Substituting D_i for d_i for those $i : w_i = 0$, ${\scriptstyle (1 \leq i \leq n)}$, gives

$$\mathcal{S}_b = \sum_{i \in I} (w_i)(\prod_{\substack{j \neq i \\ j \in I}} d_j)(\prod_{\substack{1 \leq q \leq n \\ q \notin I}} D_q).$$

Similarly, φ may be written

$$\varphi = (\prod_{j \in I} d_j)(\prod_{\substack{1 \leq q \leq n \\ q \notin I}} D_q) - \mathcal{V} = 0$$

Using the method of Lagrange multipliers, the set of partial differential equations for $l \in I$, ${\scriptstyle (0 \leq l < k)}$ is [6]

$$\frac{\partial \mathcal{S}_b}{\partial d_l} + \lambda(\frac{\partial \varphi}{\partial d_l}) = \sum_{\substack{i \neq l \\ i \in I}} (w_i)(\prod_{\substack{j \neq i,l \\ j \in I}} d_j) + \lambda(\prod_{\substack{j \neq l \\ j \in I}} d_j) = 0$$

Multiplying the l-th partial differential equation by d_l,

$$\sum_{\substack{i \neq l \\ i \in I}} (w_i)(\prod_{\substack{j \neq i \\ j \in I}} d_j) + \lambda(\prod_{j \in I} d_j) = 0.$$

This produces a set of equations E_k, such that upon subtraction of $E_m - E_{m+1}$, $(0 \leq m < k$, $m+1$ evaluated mod $k)$, all summands except those two containing $w_{I(m)}, w_{I(m+1)}$ in the product are identical.

[6] The constant product $\prod_{\substack{1 \leq q \leq n \\ q \notin I}} (D_q)$ occurs in each summand of $\mathcal{S}_b$ and in φ. Both sides of the equation are divided by this product.

The result of this subtraction is the set of equations

$$\sum_{\substack{i \neq l \\ i \in I}} (w_i)(\prod_{\substack{j \neq i \\ j \in I}} d_j) = \sum_{\substack{I(i) \neq I(l+1) \\ i \in I}} (w_i)(\prod_{\substack{j \neq i \\ j \in I}} d_j)$$

which simplifies to

$$w_{I(l+1)} \prod_{j \neq I(l+1)} d_j = w_{I(l)} \prod_{j \neq I(l)} d_j.$$

Upon further simplification, the d_j $_{(j \neq I(l), I(l+1))}$ cancel and the resulting equations are

$$w_{I(l+1)} d_{I(l)} = w_{I(l)} d_{I(l+1)}, \quad (0 \leq l < k \text{ and } (l+1) \text{ evaluated mod } k)$$

or, rewritten as ratios,

$$\frac{w_{I(l+1)}}{w_{I(l)}} = \frac{d_{I(l+1)}}{d_{I(l)}}.$$

□

Claim 2

The block dimensions are

$$\log_2 d_i = \begin{cases} \frac{1}{k} \log_2 \frac{(w_i)^k \prod_{j \in I} D_j}{P \prod_{j \in I} w_j} & \text{if } i \in I \text{ and } d_i \leq D_i \\ \log_2 D_i & \text{otherwise} \end{cases} \tag{2}$$

where $1 \leq i \leq n$.

Proof

Let $1 \leq i \leq n$, $i, j, l, (j+1), (l+1) \in I$, $j' \notin I$, and $(l+1), (j+1)$ be evaluated mod k.

In order to solve for $d_i, i \in I$, take the ratio equation 1

$$\frac{d_{I(l)}}{d_{I(l+1)}} = \frac{w_{I(l)}}{w_{I(l+1)}}$$

and solve for $d_{I(l)}$.

$$d_{I(l)} = \frac{w_{I(l)}}{w_{I(l+1)}} d_{I(l+1)} \tag{3}$$

Substituting for those $d_i = D_i$ the volume equation becomes

$$\begin{aligned}\mathcal{V} &= \prod d_i = \frac{\prod D_i}{P} \\ &= \prod_{j\in I} d_j \prod_{j'\notin I} D_{j'} = \frac{(\prod_{j\in I} D_j)(\prod_{j'\notin I} D_{j'})}{P} \\ &= \prod_{j\in I} d_j = \frac{\prod_{j\in I} D_j}{P}\end{aligned}$$

Next, rewrite $\prod_{j\in I} d_j$ as

$$(d_{I(l)})(d_{I(l+1)})(\prod_{\substack{j\in I\\ j\neq l,l+1}} d_j) = \frac{\prod_{j\in I} D_j}{P}$$

Solving for $d_{I(l+1)}$

$$d_{I(l+1)} = \frac{\prod_{j\in I} D_j}{P d_{I(l)} \prod\limits_{\substack{j\in I\\ j\neq l,l+1}} d_j}$$

and substituting into 3 gives

$$\begin{aligned}d_{I(l)} &= \frac{w_{I(l)}}{w_{I(l+1)}} \frac{\prod_{j\in I} D_j}{P d_{I(l)} \prod\limits_{\substack{j\in I\\ j\neq l,l+1}} d_j} \\ (d_{I(l)})^2 &= \frac{w_{I(l)}}{w_{I(l+1)}} \frac{\prod_{j\in I} D_j}{P \prod\limits_{\substack{j\in I\\ j\neq l,l+1}} d_j}\end{aligned}$$

Using the remaining $k-1$ ratio equations to substitute weights for dimensions, and by Lemma A the resulting solution for d_i is

$$\begin{aligned}{d_{I(l)}}^k &= \frac{(w_{I(l)})^{k-1} \prod_{j\in I} D_j}{P \prod_{\substack{j\in I\\ j\neq l}} w_j} \\ &= \frac{(w_{I(l)})^{k} \prod_{j\in I} D_j}{P \prod_{j\in I} w_j} && (4) \\ \log d_{I(l)} &= \frac{1}{k} \log \frac{(w_{I(l)})^{k} \prod_{j\in I} D_j}{P \prod_{j\in I} w_j} && (5)\end{aligned}$$

Consider d_i, $(1 \leq i \leq n)$.

1. $d_i > 0$.

   All factors of the quotient on the RHS of Equation 4 are positive, and so the quotient is positive. The k-th root of d_i is therefore positive.

2. $i \notin I$.

   Then by Lemma A $w_i = 0$ and $d_i = D_i$.

3. Suppose $d_i > D_i$.

   The equations in the problem statement specify a family of problems since they reflect the volume of the iteration space, but are indifferent to the shape of the iteration space. Therefore, it is possible to get spurious solutions, i.e., solutions that satisfy the equations but not the intended shape of the iteration space. A spurious solution is indicated when the computed d_i exceeds the iteration space dimension D_i.

   Intuitively, these equations represent a worst-case solution: the constraint on communication assumes there exists at least one block which is internal to the iteration space, viz., completely surrounded by other blocks.

   By setting $d_i := D_i$, and Lemma A, the problem is refined by reducing the dimension for communication to $k' := k-1$. This is equivalent to setting $w_i := 0$ and recomputing the index set I. Therefore, d_i is defined by Equation 2.

The computation of d_i is iterative, and guaranteed to terminate. Iteration over $d_{1 \leq k \leq n}$ is required only if $d_i > D_i$. In this case, $w_i := 0$ and $d_i := D_i$. Each iteration reduces the number of undefined block dimensions by at least one. The algorithm terminates when all d_i are defined.

$\square$.