

Generating Efficient Tiled Code for Distributed Memory Machines

Peiyi Tang[†] and Jingling Xue[‡]

Department of Mathematics and Computing
University of Southern Queensland, Toowoomba, QLD 4350, Australia[†]
School of Computer Science and Engineering
The University of New South Wales, Sydney, NSW 2351, Australia[‡]

Abstract — *Tiling can improve the performance of nested loops on distributed memory machines by exploiting coarse-grain parallelism and reducing communication overhead and frequency. Tiling calls for a compilation approach that performs first computation distribution and then data distribution, both possibly on a skewed iteration space. This paper presents a suite of compiler techniques for generating efficient SPMD programs to execute rectangularly tiled iteration spaces on distributed memory machines. The following issues are addressed: computation and data distribution, message-passing code generation, memory management and optimisations, and global to local address translation. Methods are developed for partitioning arbitrary iteration spaces and skewed data spaces. Techniques for generating efficient message-passing code for both arbitrary and rectangular iteration spaces are presented. A storage scheme for managing both local and nonlocal references is developed, which leads to the SPMD code with high locality of references. Two memory optimisations are given to reduce the amount of memory usage for skewed iteration spaces and expanded arrays, respectively. The proposed compiler techniques are illustrated using a simple running example and finally analysed and evaluated based on experimental results on a Fujitsu AP1000 consisting of 128 processors.*

Index Terms. Nested Loops, Tiling, Distributed Memory Machines, SPMD, Memory Optimisations.

1 Introduction

Distributed memory machines such as IBM SP-2, TMC CM-5 and Fujitsu AP1000 rely on message passing for interprocessor communication and synchronisation. To obtain good performance on these machines, it is important to efficiently manage communication between processors. For many applications, exploiting coarse-grain parallelism can amortise the impact of communication overhead by reducing communication overhead and frequency.

Iteration space tiling, also known as *blocking*, is an important loop optimisation technique for regularly structured computations where the most of execution time is spent in nested loops. Example applications include some dense matrix algebra and regular grid-based finite-difference and finite-element algorithms. The data dependences of these algorithms usually span the entire iteration space, making it impossible to create coarse-grain DOALL parallelism. The loop nests of this kind are known as *DOACROSS loop nests* [37].

Tiling aggregates small loop iterations into tiles. By executing tiles as atomic units of computation, communication takes place per tile instead of per iteration. By adjusting the size of tiles, tiling can achieve coarse-grain DOACROSS parallelism while reducing communication overhead and frequency on distributed memory machines [13, 26, 29, 30, 33, 38]. While a lot of work has been done on tiling or other related optimisations [10, 32], little research efforts in the literature are

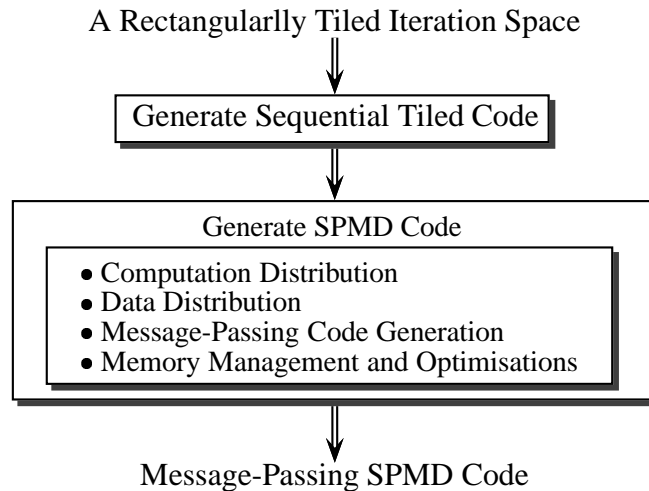


Figure 1: Generating SPMD code for tiled iteration spaces

found on automatic generation of efficient blocked code for distributed memory machines, which is the subject of this paper.

Several prototype compilers, such as Fortran-D [19] and Vienna Fortran [2], have been developed for compiling HPF-like data-parallel languages. These systems first find the data distribution (usually specified by the programmer) and then infer the computation distribution (using the owner-computes rule [19] or others [3]). When compiling a tiled iteration space, however, the proposed approach will carry out these two phases in the reverse order. This is because once the iteration space is tiled, the computation distribution is partially determined. According to the so-called atomic tiles constraint [20, 39], each tile is an atomic unit of work to be scheduled on a processor, and once scheduled, a tile runs to completion without preemption. Therefore, when generating code for tiled iteration spaces, it is natural to determine first the computation distribution and then the data distribution.

This paper is concerned with compiling tiled iteration spaces for efficient execution on distributed memory machines. Given a rectangularly tiled iteration space, this paper provides a suite of compiler techniques for generating a message-passing SPMD program to be run on each processor. Our compilation approach is illustrated in Figure 1. In our SPMD execution mode, all processors execute the same program but on different data sets, and in addition, each processor computes only the values of data it owns. This means that the owner-computes rule [19] is still enforced in our generated SPMD programs, although it is not used in deriving the computation distribution. We introduce compiler techniques to solve the following problems: computation distribution, data distribution, message-passing code generation, memory management, memory optimisations, and global to local address translation. By exploiting the regularity of data dependences, both the proposed compiler techniques and the generated SPMD code are simple and efficient.

This paper contains the following main results. First, we propose methods for deriving the computation and data distribution for a tiled iteration space. The iteration space is assumed to be convex and of any arbitrary shape. The tiles are distributed cyclically in some dimensions and collapsed

in the remaining dimensions. This tends to achieve good load balance for non-rectangular iteration spaces and overlap computation and communication. By adjusting the tile size, the general block-cyclic distribution can be implemented. Given the computation distribution, the data distribution is found using the “computer-owns” rule, by which a processor owns the data it computes. For skewed iteration spaces, an array is partitioned in a skewed block-cyclic fashion. Such data distribution cannot be specified by HPF data mapping directives [22].

Second, we present techniques for generating efficient message-passing code for both arbitrary and rectangular iteration spaces. The problem is nontrivial when iteration spaces have arbitrary shapes and when empty tiles can be enumerated. In our communication scheme, small messages are aggregated into large messages, amortising the large startup overhead on distributed memory machines. In addition, each message comprises the elements from a regular array section of an array, yielding efficient code for packing and unpacking messages.

Third, we describe a memory compression technique for allocating local memory for distributed arrays. Both the local and nonlocal data of a distributed array are stored in the same local array. The nonlocal data are stored using an extended concept of overlap areas [15]: they are organised according to the loop indices at which they are computed rather than their array indices. This storage scheme leads to high locality of references for the computations within tiles regardless of any sparse array references that may appear in the original program. We provide various formulas for translating between global and local addresses. We generate efficient SPMD code by avoiding unnecessary division and modulo operations.

Fourth, we have developed two optimisation techniques to reduce the amount of memory usage. In particular, *address folding* is used to deal with skewed iteration spaces and *memory recycling* with expanded arrays. If the iteration space is rectangular and then skewed, the address folding ensures that no extra memory will be allocated. In the case of an expanded array, the extra memory needed in an expanded dimension is increased only by the size of the tile (not the size of the array) along that dimension.

Finally, we have validated and evaluated all proposed compiler techniques. We present some experimental results based on the SOR example and identify some performance-critical factors that impact the performance of tiled code on distributed memory machines.

The rest of this paper is organised as follows. Section 2 provides the background information for the paper. In particular, we describe our approach to generating sequential tiled code given a tiled iteration space. We present our compiler techniques in Sections 3 – 8, with an emphasis on communication code generation and memory management. In Section 9, we evaluate our compiler techniques based on our experiments on an Fujitsu AP1000 and identify future research areas for improving the performance of blocked code. Section 10 discusses related work, and finally the paper is concluded in Section 11.

2 Background

2.1 Notation and Terminology

A single up-case letter in **bold** font is used to represent a set. The notation \vec{x} stands for a vector, and $\vec{x}[i : j]$ denotes the subvector of \vec{x} containing the entries x_i, \dots, x_j in that order. $\vec{0}$ and $\vec{1}$ denote

all-zero and all-one vectors, respectively. If $\vec{a} = (a_1, \dots, a_n)$ and $\vec{b} = (b_1, \dots, b_n)$, we define $\vec{a} \circ \vec{b} = (a_1 b_1, \dots, a_n b_n)$. The integer modulo operator mod is defined such that $a \text{ mod } b = a - b \lfloor \frac{a}{b} \rfloor$. Thus, $a \text{ mod } b \geq 0$ whenever $b > 0$. The operators such as $/$ and mod on two vectors are component-wise. As is customary, $<$ and \leq represent the lexicographic order “less than” and “less than or equal to”, respectively. Let $\mathbf{S} \subset \mathbb{Z}^n$. We define $\min_{<} \mathbf{S} = \{\vec{x} \in \mathbf{S} \mid \forall \vec{y} \in \mathbf{S} : \vec{x} \neq \vec{y} \implies \vec{x} < \vec{y}\}$.

2.2 Program Model

Tiling focuses on very structured computations expressed in perfectly nested loops with constant data dependences. Without complicating the presentation unduly, we consider a simple program model defined as follows:

$$\begin{aligned} & \text{for}(i_1 = L_1; i_1 \leq U_1; i_1++) \\ & \quad \dots \\ & \quad \text{for}(i_n = L_n; i_n \leq U_n; i_n++) \\ & \quad \quad A(f(\vec{i})) = F(A(f(\vec{i} - \vec{d}_1)), \dots, A(f(\vec{i} - \vec{d}_r))); \end{aligned}$$

However, with some notational extensions (by adding more subscripts, for example), all the compiler techniques proposed in the paper can be used in the case of multiple array variables appearing in multiple statements, each of which may possibly be guarded by conditionals. The only real restriction is that the program must have constant dependence vectors.

It is assumed that all dependences are flow (i.e., true) dependences, implying that the program has single assignment semantics. Methods for removing anti and output dependences include array expansion [14], node splitting [27], array privatisation [17], and others [4]. The absence of output dependences makes it easy to enforce the owner-computes rule when the computation distribution is performed before the data distribution. For reasons of symmetry, the compiler techniques we propose for dealing with flow dependences can be used to deal with anti dependences as well. But anti dependences can be satisfied by collective communication faster than point-to-point communication.

Many stencil-based programs like SOR consist of an outermost time step that performs iteratively the computations described by the inner loops. The output data dependences in these programs can be removed simply by array expansion [14]. We use a memory recycling technique to minimise the amount of memory usage for expanded arrays (Section 9).

In our program model, all loops have stride 1 and the loop bounds L_k and U_k are affine expressions of outer loop variables i_1, \dots, i_{k-1} . Thus, the *iteration space* is defined by:

$$\mathbf{I} = \{\vec{i} \mid \forall k : L_k \leq i_k \leq U_k\}$$

where $\vec{i} = (i_1, \dots, i_n)$ denotes an iteration vector. Each read reference has the form $A(f(\vec{i} - \vec{d}))$, where $\vec{d} \in \mathbb{Z}^n$ is a constant dependence vector. The subscript function f is an affine expression of loop variables; it is injective over the iteration space \mathbf{I} because the program has single assignment semantics. \mathbf{D} denotes the set of all dependence vectors in the program: $\mathbf{D} = \{\vec{d}_1, \dots, \vec{d}_r\}$. Because we deal with DOACROSS loop nests, the dependence vectors are assumed to span the entire iteration space, i.e., $\text{span}(\mathbf{D}) = \mathbb{Z}^n$. (This implies that $r \geq n$.)

The following program is used as a running example to illustrate various compiler techniques used in the entire compilation process.

EXAMPLE 1 Consider the following double loop:

```

for( $i = 1; i \leq 9; i++$ )
  for( $j = 1; j \leq 4; j++$ )
     $A(i, 2j) = A(i, 2j - 2) + A(i - 1, 2j - 2)$ 

```

By putting the statement into the following form:

$$A \left(\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \right) = A \left(\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \left(\begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) \right) + A \left(\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \left(\begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \right)$$

we can see easily that all data dependences in the program are constant: $\mathbf{D} = \{(0, 1), (1, 1)\}$, and the subscript function f , defined as $f(\begin{bmatrix} x \\ y \end{bmatrix}) = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$, is injective over the iteration space $\mathbf{I} = \{(i, j) \mid 1 \leq i \leq 9 \wedge 1 \leq j \leq 4\}$ depicted in Figure 2(a).

2.3 Iteration Space Tilings

Section 2.3.1 defines a rectangular tiling as a loop transformation from \mathbb{Z}^n to \mathbb{Z}^{2n} . Section 2.3.2 calculates the dependences between tiles useful for generating message-passing code. Section 2.3.3 focuses on generating the sequential tiled code, which will be refined gradually into an SPMD program.

2.3.1 Rectangular Tilings

Tiling divides the iteration space into uniform tiles and traverses the tiles to cover the entire iteration space. For example, as shown in Figure 2(b), the iteration space in Figure 2(a) has been divided into 10 tiles, each of which is a 2×2 rectangle. In general, the size of a tile can be characterised by an n -dimensional vector, called the *tile size vector*, $\vec{B} = (B_1, \dots, B_n)$, where B_k is the number of iterations in dimension k . The number of iterations in a *full* tile is, thus, $B_1 \times \dots \times B_n$, but the number of iterations can be less for tiles at the boundaries of the iteration space. In Figure 2(b), a full tile contains $2 \times 2 = 4$ iterations, but the two boundary tiles each contain two iterations only.

To find the mapping from iterations to nonnegative tile indices, we need the largest integer vector \vec{i}_{\min} such that $\vec{i}_{\min} \leq \vec{i}$ for all iterations \vec{i} in the iteration space. That is, $i_{\min,k} = \min(i_k \mid (i_1, \dots, i_n) \in \mathbf{I})$. This vector can be found using Fourier-Motzkin elimination [31, p. 49] or the PIP system [12]. For the iteration space in Example 1, we have $\vec{i}_{\min} = (1, 1)$.

A rectangular tiling on an n -dimensional iteration space can be modeled as a mapping to a $2n$ -dimensional tiled iteration space defined as follows:

$$\rho : \mathbf{I} \mapsto \mathbf{I}', \quad \rho(\vec{i}) = \begin{pmatrix} \vec{t} \\ \vec{e} \end{pmatrix} = \begin{pmatrix} \lfloor \frac{\vec{i} - \vec{i}_{\min}}{\vec{B}} \rfloor \\ (\vec{i} - \vec{i}_{\min}) \bmod \vec{B} \end{pmatrix} \quad (1)$$

where the *tiled iteration space* is the range of the mapping: $\mathbf{I}' = \{\rho(\vec{i}) \mid \vec{i} \in \mathbf{I}\}$. The first component $\vec{t} = \lfloor \frac{\vec{i} - \vec{i}_{\min}}{\vec{B}} \rfloor$ identifies the index of the tile that contains the iteration \vec{i} and the second component $\vec{e} = (\vec{i} - \vec{i}_{\min}) \bmod \vec{B}$ gives the position of \vec{i} relative to the tile origin. The *origin* of tile \vec{t} is the

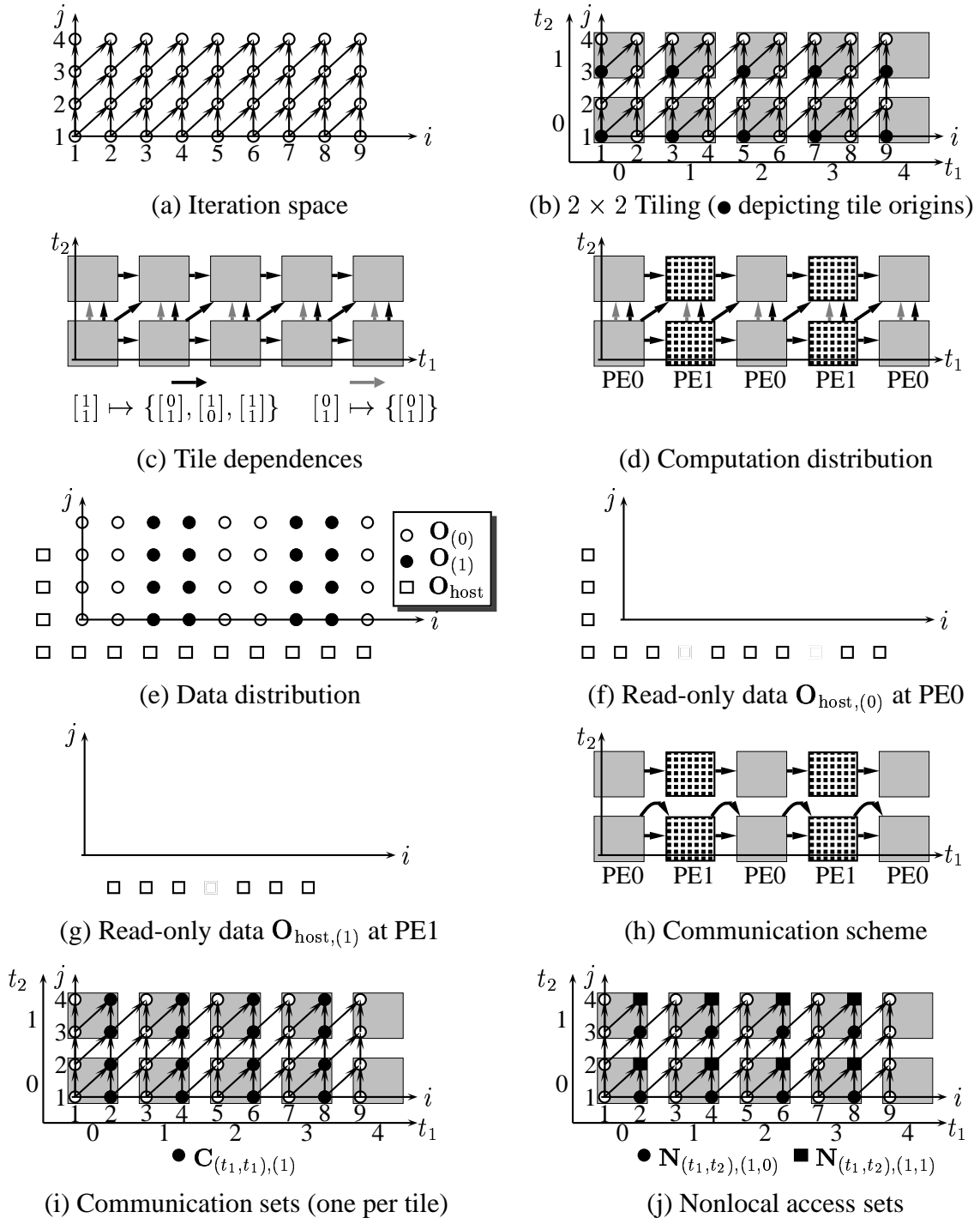


Figure 2: A running example.

unique iteration \vec{i} such that $\vec{i} - \vec{i}_{\min} = \vec{t} \circ \vec{B}$. As an example, the origins of all tiles in Figure 2(b) are depicted in solid circles. Our definition of mod ensures that $\vec{e} = (\vec{i} - \vec{i}_{\min}) \bmod \vec{B} \geq 0$.

The *tile space*, denoted \mathbf{T} , is the set of all tile index vectors, which is the projection of the tiled iteration space over its first n coordinates:

$$\mathbf{T} = \left\{ \left\lfloor \frac{\vec{i} - \vec{i}_{\min}}{\vec{B}} \right\rfloor \mid \vec{i} \in \mathbf{I} \right\}$$

The tile space for the example in Figure 2 is $\mathbf{T} = \{(t_1, t_2) \mid 0 \leq t_1 \leq 4 \wedge 0 \leq t_2 \leq 1\}$.

A rectangular tiling is an iteration-reordering transformation. To be legal, all dependence vectors must be nonnegative [20, 39]. If a loop nest has dependence vectors with negative entries, the loop nest can be skewed to make all dependence vectors nonnegative [37]. The skewed loop nest can then be tiled by rectangles in the normal manner.

The iteration space can be directly tiled legally with parallelepipeds [20, 39]. However, in most practical cases, a parallelepiped tiling consists of a skewing followed by a rectangular tiling. Our compiler techniques can then be applied to the skewed iteration space.

In the following discussions, all dependence vectors are assumed to be nonnegative.

2.3.2 Dependences between Tiles

To generate message-passing code, it is necessary to find out the data dependences between tiles called the *tile dependences*. A tile depends on another tile if there exists a path of data dependences from the latter to the former. In general, the tile dependences can be calculated from the data dependences in the program and the tile size used [39]. Unlike a unimodular transformation, a tiling transformation can transform a single dependence vector between iterations into several tile dependence vectors. Precisely, the dependence vector $\vec{d} = (d_1, \dots, d_n) \in \mathbf{D}$ is mapped to the following set of tile dependence vectors:

$$\delta : \mathbf{D} \mapsto 2^{\mathbf{Z}^n}, \delta \left(\begin{bmatrix} d_1 \\ \vdots \\ d_n \end{bmatrix} \right) = \left\{ \begin{bmatrix} d'_1 \\ \vdots \\ d'_n \end{bmatrix} \neq 0 \mid d'_k \in \left\{ \lfloor \frac{d_k}{B_k} \rfloor, \lceil \frac{d_k}{B_k} \rceil \right\} \right\}$$

Let Δ be the set of all tile dependence vectors. We have:

$$\Delta = \cup_{\vec{d} \in \mathbf{D}} \delta(\vec{d}) \quad (2)$$

As illustrated in Figure 2(c), $\delta((0, 1)) = \{(0, 1)\}$ and $\delta((1, 1)) = \{(0, 1), (1, 0), (1, 1)\}$. So the dependence vector $(1, 1)$ alone induces the dependences with all its neighbouring tiles. Thus, $\Delta = \{(0, 1), (1, 0), (1, 1)\}$.

For practical nested loops with constant dependences, the tile size is sufficiently larger than the magnitudes of distance vectors [16]. We assume $\forall k : d_k \leq B_k$ holds for every dependence vector \vec{d} in the program. This leads to the following theorem.

THEOREM 1 *If $(\forall \vec{d} \in \mathbf{D} : \vec{d} \leq \vec{B})$, then $\Delta \subseteq (\{0, 1\}^n \setminus \vec{0})$.*

Let us apply this theorem to two extreme cases. If \mathbf{D} contains only the n elementary basis vectors, i.e., $\mathbf{D} = \{(1, 0, \dots, 0), (0, 1, 0, \dots, 0), \dots, (0, \dots, 0, 1)\}$, then $\Delta = \mathbf{D}$. If $(1, \dots, 1) \in \mathbf{D}$, then $\Delta = \{0, 1\}^n - \vec{0}$.

2.3.3 Sequential Tiled Code

Once a tiling transformation is found, we need to generate a sequential tiled program to execute the points in the tiled iteration space lexicographically. This program serves as the basis from which the final SPMD program is gradually derived. Thus, we want the sequential code to be simple and efficient so that the SPMD code can also be simple and efficient.

The sequential tiled code consists of $2n$ loops that are perfectly nested, where the outer n *tile loops* step between tiles and the inner n *element loops* enumerate the iterations within a tile. The tile loops are constructed using Fourier-Motzkin elimination [31] or its extensions [12, 28].

According to the tiling transformation ρ given in (1), all iterations that satisfy

$$\vec{B} \circ \vec{t} \leq \vec{i} - \vec{i}_{\min} \leq \vec{B} \circ \vec{t} + \vec{B} - \vec{1}$$

are mapped to the same tile \vec{t} . Thus, the set of iterations in the tile can be defined by:

$$\mathbf{I}_{\vec{t}} = \{\vec{i} \in \mathbf{I} \mid \vec{B} \circ \vec{t} \leq \vec{i} - \vec{i}_{\min} \leq \vec{B} \circ \vec{t} + \vec{B} - \vec{1}\} \quad (3)$$

Then, the tile space can be expressed as:

$$\mathbf{T} = \{\vec{t} \mid \exists \vec{i} \in \mathbf{I} : \vec{t} \circ \vec{B} \leq \vec{i} - \vec{i}_{\min} \leq \vec{B} \circ \vec{t} + \vec{B} - \vec{1}\} \quad (4)$$

The above inequalities define a $2n$ -dimensional convex polyhedron in terms of tile indices t_1, \dots, t_n and loop indices i_1, \dots, i_n . By eliminating successively the loop indices i_1, \dots, i_n from these inequalities using Fourier-Motzkin elimination [31, p. 49] or other extensions [12, 28], we obtain a convex polyhedron defined as follows:

$$\mathbf{T}' = \{(t_1, \dots, t_n) \mid \forall k : L_k^t \leq t_k \leq U_k^t\}$$

where L_k^t and U_k^t are affine expressions of t_1, \dots, t_{k-1} and problem size parameters and can be simplified as follows when the iteration space is rectangular:

$$\begin{aligned} L_k^t &= 0 \\ U_k^t &= \lfloor \frac{U_k - L_k}{B_k} \rfloor \end{aligned} \quad (5)$$

These inequalities give rise to the tile loops as shown in Figure 3. Thus \mathbf{T}' is the iteration space of the tile loops and will be called the *effective tile space*. By construction, $\mathbf{T}' \supseteq \mathbf{T}$ but $\mathbf{T}' = \mathbf{T}$ may not be true. As a result, some *empty tiles* (containing no iterations) may be enumerated by the tile loops.

The tile loops are said to be *exact* if $\mathbf{T} = \mathbf{T}'$, i.e., if no empty tiles are enumerated.

The element loops shown in Figure 3 are obtained straightforwardly. Note that the original loop bounds L_k and U_k are used in the loop bounds of these element loops to ensure that only the iterations in the original iteration space are executed; they are redundant when all tiles are full tiles.

In general, the tile space \mathbf{T} is not a convex polyhedron. Thus, the tile loops may be inexact. While it is possible to generate multiple loop nests to enumerate the tile space exactly [28], one single perfect loop nest is preferred for several reasons. First, our tile loops are expected to be exact for real programs. Second, the message passing code and its generation are simple. Third, the final

```

for( $t_1 = L_1^t; t_1 \leq U_1^t; t_1++$ )
  ...
  for( $t_n = L_n^t; t_n \leq U_n^t; t_n++$ )
    for( $i_1 = \max(L_1, t_1 B_1 + i_{\min,1}); i_1 \leq \min(U_1, (t_1 + 1)B_1 - 1 + i_{\min,1}); i_1++$ )
      ...
      for( $i_n = \max(L_n, t_n B_n + i_{\min,n}); i_n \leq \min(U_n, (t_n + 1)B_n - 1 + i_{\min,n}); i_n++$ )
         $A(f(\vec{i})) = F(A(f(\vec{i} - \vec{d}_1)), \dots, A(f(\vec{i} - \vec{d}_r)));$ 

```

Figure 3: Sequential tiled code.

SPMD code derived from the tiled code is simple. We must emphasise that although $\mathbf{T} = \mathbf{T}'$ cannot be guaranteed and the tile loops in Figure 3 may traverse empty tiles, the iterations of the original program traversed by the element loops are exact.

The tiled code for the rectangular tiling in Figure 2(b) can be obtained as follows:

```

for( $t_1 = 0; t_1 \leq 4; t_1++$ )
  for( $t_2 = 0; t_2 \leq 1; t_2++$ )
    for( $i = \max(1, 2t_1 + 1); i \leq \min(9, 2t_1 + 2); i++$ )
      for( $j = \max(1, 2t_2 + 1); j \leq \min(4, 2t_2 + 2); j++$ )
         $A(i, 2j) = A(i, 2j - 2) + A(i - 1, 2j - 2)$ 

```

(6)

where the lower bounds of i and j can be simplified to $2t_1 + 1$ and $2t_2 + 1$ and the upper bound of j to 4, respectively.

2.4 Machine Model

Parallel processors of a distributed memory machine are arranged as an m -dimensional mesh:

$$\mathbf{P} = \{\vec{0} \leq \vec{p} < \vec{P}\}$$

where $\vec{P} = (P_1, \dots, P_m)$ is a positive integer vector such that $P_k > 1$ represents the number of processors in the k -th dimension. Each processor is uniquely identified by a processor index vector $\vec{p} \in \mathbf{P}$. The total number of processors in the processor mesh is $P_1 \times \dots \times P_m$. Each processor may directly communicate with any processor in the target machine via message passing.

3 Computation Distribution

The computation distribution is to determine which tiles are computed at which processors. Because a tile is an atomic unit of computation, all iterations in the same tile are computed at the same processor.

The tiles are allocated to processors cyclically according to the *tile allocation function*:

$$\phi : \mathbf{T} \mapsto \mathbf{P} : \phi(\vec{t}) = \vec{t}[1 : m] \bmod \vec{P} \quad (7)$$

This can be specified in HPF as $\overbrace{(\mathbf{cyclic}(1), \dots, \mathbf{cyclic}(1), *, \dots, *)}^m$ [22]. That is, the first m dimensions of the tile space are distributed cyclically to processors and the last $n - m$ are collapsed. The iterations in the k -th distributed dimension of the iteration space are distributed using $\mathbf{cyclic}(B_k)$, where B_k is the tile size in the k -th dimension.

Since the loops are fully permutable when all dependence vectors are nonnegative [36]. Any combination of m dimensions can be distributed once the corresponding loops are permuted to become the outer m loops.

Let $\mathbf{T}_{\vec{p}}$ be the set of tiles allocated to processor \vec{p} . We have:

$$\mathbf{T}_{\vec{p}} = \{\vec{t} \mid \vec{t} \in \mathbf{T} \wedge \phi(\vec{t}) = \vec{p}\}$$

Based on (4), $\mathbf{T}_{\vec{p}}$ can be defined by the following set of linear inequalities:

$$\mathbf{T}_{\vec{p}} = \{\vec{t} \mid \exists \vec{i} \in \mathbf{I} : \vec{t} \circ \vec{B} \leq \vec{i} - \vec{i}_{\min} \leq \vec{B} \circ \vec{t} + \vec{B} - \vec{1} \wedge \vec{t} \bmod \vec{P} = \vec{p}\}$$

where the modulo constraint can be made linear by introducing extra variables, a standard technique used in HPF compilers [11].

The iterations allocated to processor \vec{p} are the iterations contained in all tiles that are allocated to that processor. The portion of the iteration space allocated to the processor \vec{p} , called the *local iteration space* and denoted $\mathbf{I}_{\vec{p}}$, is defined by the same inequalities as $\mathbf{T}_{\vec{p}}$:

$$\mathbf{I}_{\vec{p}} = \{\vec{i} \in \mathbf{I} \mid \exists \vec{t} : \vec{t} \circ \vec{B} \leq \vec{i} - \vec{i}_{\min} \leq \vec{B} \circ \vec{t} + \vec{B} - \vec{1} \wedge \vec{t} \bmod \vec{P} = \vec{p}\}$$

Suppose the tiles in Figure 2(c) are distributed over a linear arrangement of two processors. That is, $\mathbf{P} = \{(0), (1)\}$ and $\vec{P} = (2)$. Then the first dimension of the tile space is cyclically distributed to the two processors as shown in Figure 2(d) in that dimension, and the tiles in the other dimension are all allocated to the same processor. The tiles allocated to processor $\vec{p} = (p_1)$, where $0 \leq p_1 \leq 1$, can be calculated as:

$$\mathbf{T}_{(p_1)} = \{(t_1, t_2) \mid \exists i, j : \max(1, 2t_1 + 1) \leq i \leq \min(9, 2t_1 + 2) \wedge \max(1, 2t_2 + 1) \leq j \leq \min(4, 2t_2 + 2) \wedge t_1 \bmod 2 = p_1\}$$

The local iteration spaces for PE0 and PE1 are specified by the same inequalities as above:

$$\mathbf{I}_{(p_1)} = \{(i, j) \mid \exists t_1, t_2 : \max(1, 2t_1 + 1) \leq i \leq \min(9, 2t_1 + 2) \wedge \max(1, 2t_2 + 1) \leq j \leq \min(4, 2t_2 + 2) \wedge t_1 \bmod 2 = p_1\} \quad (8)$$

In both cases, $t_1 \bmod 2 = p_1$ can be rewritten as $t_1 = 2c + p_1 \wedge c \geq 0$, where c is a new variable.

By distributing tiles to processors cyclically in m -dimensions, we tend to minimise load imbalance and maximise parallelism. This is especially significant for skewed iteration spaces as confirmed by our experimental results. By collapsing tiles in the remaining dimensions, the computation in some processors can be overlapped with the communication that takes place in some other processors. Furthermore, since the entries of tile dependence vectors are 0 or 1 (Theorem 1), our computation distribution requires only nearest-neighbour communication between processors.

```

    Receive and Unpack the Read-Only Data From the Host ;
for( $t_1 = L_1^t + (p_1 - L_1^t) \bmod P_1$ ;  $t_1 \leq U_1^t$ ;  $t_1 += P_1$ )
...
    for( $t_m = L_m^t + (p_m - L_m^t) \bmod P_m$ ;  $t_m \leq U_m^t$ ;  $t_m += P_m$ )
        for( $t_{m+1} = L_{m+1}^t$ ;  $t_{m+1} \leq U_{m+1}^t$ ;  $t_{m+1} ++$ )
            ...
                for( $t_n = L_n^t$ ;  $t_n \leq U_n^t$ ;  $t_n ++$ )
                    Receive and Unpack Messages for Tiles from Neighbouring PEs ;
                    for( $i_1 = \max(L_1, t_1 B_1 + i_{\min,1})$ ;  $i_1 \leq \min(U_1, (t_1 + 1) B_1 - 1 + i_{\min,1})$ ;  $i_1 ++$ )
                        ...
                            for( $i_n = \max(L_n, t_n B_n + i_{\min,n})$ ;  $i_n \leq \min(U_n, (t_n + 1) B_n - 1 + i_{\min,n})$ ;  $i_n ++$ )
                                 $A(f(\vec{i})) = F(A(f(\vec{i} - \vec{d}_1)), \dots, A(f(\vec{i} - \vec{d}_r)))$ ;
                                    Pack and Send Messages for Tiles to Neighbouring PEs ;
                                Pack and Send the Result Back to the Host ;

```

Figure 4: SPMD code for processor \vec{p} (after computation distribution).

Once tiles are partitioned across the processors, the first m tile loops in Figure 3 are modified, as shown in Figure 4, so that each processor computes its own share of tiles. For loop nests with rectangular iteration spaces, all the modulo operations are redundant because $L_k^t = 0$. The message-passing code generation as indicated in the four boxes is discussed in Sections 4 and 5.

In the case of our running example, if the tiles are distributed to PE0 and PE1 as shown in Figure 2(d), the outer tile loop given in (6) can be modified as indicated below:

$$\begin{aligned}
 & \text{for}(t_1 = p_1; t_1 \leq 4; t_1 += 2) \\
 & \quad \text{for}(t_2 = 0; t_2 \leq 1; t_2 ++) \\
 & \quad \quad \text{for}(i = \max(1, 2t_1 + 1); i \leq \min(9, 2t_1 + 2); i ++) \\
 & \quad \quad \quad \text{for}(j = \max(1, 2t_2 + 1); j \leq \min(4, 2t_2 + 2); j ++)
 \end{aligned} \tag{9}$$

so that each processor computes only the tiles it is allocated to.

4 Computer-Owns Rule, Data Distribution and I/O Code Generation

The data distribution is to determine the home processors where the data elements of an array are *permanently* stored. Given a computation distribution, the data distribution is found using the so-called *computer-owns* rule, by which a processor owns the data it computes. This rule has two desirable implications. First, the owner-computes rule is still enforced in our SPMD programs because the program has single assignment semantics. Second, the read-only data of an array are not owned by any slave processors but owned by the host (i.e., master) processor. We describe below the inequalities that specify the read-only data owned by the host and the computed data owned by

each processor. These inequalities are used to generate the message-passing code to perform the I/O as indicated in the top and bottom boxes in Figure 4. For single assignment programs, only the read-only data are required to be sent initially from the host to the slave processors.

The data allocated to a particular processor \vec{p} are the data computed by that processor. The portion of the data space of the array A allocated to processor \vec{p} , called the *local data space* and denoted $\mathbf{O}_{\vec{p}}$, can be calculated as follows:

$$\mathbf{O}_{\vec{p}} = \{f(\vec{i}) \mid \vec{i} \in \mathbf{I}_{\vec{p}}\} \quad (10)$$

This rule of data distribution is called the *computer-owns* rule, because an element is owned by the processor that computes its value. Due to the single assignment assumption, we have $\mathbf{O}_{\vec{p}_1} \cap \mathbf{O}_{\vec{p}_2} = \emptyset$ for any two processors \vec{p}_1 and \vec{p}_2 since $\mathbf{I}_{\vec{p}_1} \cap \mathbf{I}_{\vec{p}_2} = \emptyset$. This implies that the owner-computes rule is implicitly enforced.

Let \mathbf{O}_{host} be the set of the read-only data accessed by all processors and $\mathbf{O}_{\text{host},\vec{p}}$ be the set of the read-only data accessed at the processor \vec{p} , respectively:

$$\begin{aligned} \mathbf{O}_{\text{host}} &= \{f(\vec{i} - \vec{d}) \mid \vec{i} \in \mathbf{I} \wedge (\exists \vec{d} \in \mathbf{D} : \vec{i} - \vec{d} \notin \mathbf{I})\} \\ \mathbf{O}_{\text{host},\vec{p}} &= (\cup_{\vec{d} \in \mathbf{D}} \{f(\vec{i} - \vec{d}) \mid \vec{i} \in \mathbf{I}_{\vec{p}}\}) \cap \mathbf{O}_{\text{host}} \end{aligned} \quad (11)$$

Figures 2(e) – (g) depict the data distribution and the read-only data accessed at two processors for our running example. The data owned by processor (p_1) are in the set $\mathbf{O}_{(p_1)} = \mathbf{I}_{(p_1)}$, where $\mathbf{I}_{(p_1)}$ is given in (8). The inequalities for specifying \mathbf{O}_{host} , $\mathbf{O}_{\text{host},(0)}$ and $\mathbf{O}_{\text{host},(1)}$ are omitted.

Given $\mathbf{O}_{\text{host},\vec{p}}$, we use the Omega Calculator [28] to generate the top code section in Figure 4. The host uses the same loop structure to send the corresponding read-only data to processors. For our running example, the code in the global address space for receiving the read-only data at processor (p_1), where $0 \leq p_1 \leq 1$, is:

```

if (p1 == 0)
  for(j = 0; j <= 3; j++)
    /* receive A(0,2j) */
if (p1 >= 0 && p1 <= 1)
  for(i = 0; i <= 9; i++) {
    if (-2*p1+i-2 <= 4*⌊ $\frac{-2*p_1+i-1}{4}$ ⌋)
      /* receive A(i,0) */
    if ((-2*p1+i) mod 4) == 0
      /* receive A(i,0) */
  }

```

The bottom code section in Figure 4 is derived similarly based on $\mathbf{O}_{\vec{p}}$.

If the iteration space is skewed, the data space for an array is also skewed. In this case, the data distribution cannot be specified by HPF's block or cyclic data mapping directives [22]. But the data distribution can be understood as being block-cyclically distributed on a skewed data space

5 Interprocessor Communication Code Generation

This section focuses on implementing the middle two code sections in Figure 4. That is, we discuss how to generate message-passing code to communicate the required data between processors in

order to satisfy all cross-processor dependences between tiles. Given a processor, all read-only data accessed by the processor are received initially from the host, and therefore, only the data computed nonlocally and accessed at the processor need to be communicated.

Let \mathbf{L} be the projection of Δ over the m distributed dimensions:

$$\mathbf{L} = \{\delta[1 : m] \neq \vec{0} \mid \delta \in \Delta\}$$

where every $\vec{\ell} \in \mathbf{L}$, called a *data link*, indicates that two processors \vec{p}_1 and \vec{p}_2 such that $\vec{p}_1 - \vec{p}_2 = \pm \vec{\ell}$ will communicate to each other. In general, a processor \vec{p} sends messages to the processors in $\{(\vec{p} + \vec{\ell}) \bmod \vec{P} \mid \vec{\ell} \in \mathbf{L}\}$ and receives messages from the processors in $\{(\vec{p} - \vec{\ell}) \bmod \vec{P} \mid \vec{\ell} \in \mathbf{L}\}$.

Many communication schemes are possible depending on how messages are merged. Their development and understanding can be facilitated by considering a generic tile \vec{t} and the two sets:

$$\begin{aligned} \text{succ}(\vec{t}, \vec{\ell}) &= \{\vec{t} + \vec{\delta} \mid \forall \vec{\delta} \in \Delta : \vec{\delta}[1 : m] = \vec{\ell}\} \\ \text{pred}(\vec{t}, \vec{\ell}) &= \{\vec{t} - \vec{\delta} \mid \forall \vec{\delta} \in \Delta : \vec{\delta}[1 : m] = \vec{\ell}\} \end{aligned}$$

Recall that the tile \vec{t} is allocated to the processor $\phi(\vec{t})$. The tiles, called the *successor tiles*, in $\text{succ}(\vec{t}, \vec{\ell})$ are allocated to the processor $(\phi(\vec{t}) + \vec{\ell}) \bmod \vec{P}$ and consume the data produced in \vec{t} . Reciprocally, the tiles, called the *predecessor tiles*, in $\text{pred}(\vec{t}, \vec{\ell})$ are allocated to the processor $(\phi(\vec{t}) - \vec{\ell}) \bmod \vec{P}$ and produce the data consumed by \vec{t} . The two sets for the running example are:

$$\begin{aligned} \text{succ}((t_1, t_2), (1)) &= \{(t_1 + 1, t_2), (t_1 + 1, t_2 + 1)\} \\ \text{pred}((t_1, t_2), (1)) &= \{(t_1 - 1, t_2), (t_1 - 1, t_2 - 1)\} \end{aligned}$$

Note that for the running example $\mathbf{L} = \{(1)\}$ because $\Delta = \{(0, 1), (1, 0), (1, 1)\}$.

The following two communication schemes have been evaluated and compared:

1. The data produced in the predecessor tiles $\text{pred}(\vec{t}, \vec{\ell})$ and consumed in tile \vec{t} are combined and sent in one message. This implies that the data produced in tile \vec{t} and consumed in the successor tiles in $\text{succ}(\vec{t}, \vec{\ell})$ will be sent in separate messages. As a result, the send code is more complex than the receive code. This scheme is feasible because \vec{t} cannot be computed unless all the predecessor tiles in $\text{pred}(\vec{t}, \vec{\ell})$ have been computed.
2. The data produced in tile \vec{t} and consumed in the successor tiles in $\text{succ}(\vec{t}, \vec{\ell})$ are combined and sent in one message. This implies that the data produced in the predecessor tiles in $\text{pred}(\vec{t}, \vec{\ell})$ and consumed in tile \vec{t} will be received in separate messages. As a result, the receive code is more complex than the send code.

Both schemes naturally encompasses the three classic optimisations: message vectorisation, coalescing and aggregation [19]. Both schemes are the mirror image of each other. However, we have adopted the second scheme because the property of sending the data produced in a tile in one single message can be exploited to reduce memory cost for expanded arrays (Section 8.2).

Figure 2(h) illustrates the second scheme, i.e., the communication scheme presented in the paper. Take tile $(0, 0)$ allocated to PE0 for example. The two data elements depicted in solid dots are accessed in the two tiles $(1, 0)$ and $(1, 1)$ allocated to PE1. PE0 will send to PE1 the two elements

in one message at the completion of tile $(0, 0)$. PE1 will not start executing tile $(1, 0)$ until it has received this message from PE0.

In Section 5.1, we derive the inequalities that specify the data to be communicated between processors. In Section 5.2, we discuss the generation of the required message-passing code. Section 5.3 contains simplified message-passing code for rectangular iteration spaces.

5.1 Communication Sets

In our communication scheme, the communication set represents the set of data produced in one tile at one processor and consumed by all the tiles in another processor. Precisely, the *communication set* $\mathbf{C}_{\vec{t}, \vec{\ell}}$ is the set of the data produced in tile \vec{t} at processor $\phi(\vec{t})$ and consumed by all tiles in $\text{succ}(\vec{t}, \vec{\ell})$ at the processor $(\phi(\vec{t}) + \vec{\ell}) \bmod \vec{P}$. Both the sending and receiving processor identifiers are implicit in the notation $\mathbf{C}_{\vec{t}, \vec{\ell}}$.

Each tile \vec{t} generates exactly $|\mathbf{L}|$ communication sets, where $|\mathbf{L}| \leq 2^m - 1$.

If we use $\mathbf{N}_{\vec{t}, \vec{\delta}}$ to represent the set, called the *nonlocal access set*, of the data produced in tile \vec{t} and consumed in tile $\vec{t} + \vec{\delta}$, then $\mathbf{C}_{\vec{t}, \vec{\ell}}$ is simply the union of the nonlocal access sets corresponding to all tiles in the successor set $\text{succ}(\vec{t}, \vec{\ell})$ defined below:

$$\mathbf{C}_{\vec{t}, \vec{\ell}} = \bigcup_{\forall \vec{\delta} \in \Delta: \vec{\delta}[1:m] = \vec{\ell}} \mathbf{N}_{\vec{t}, \vec{\delta}} \quad (12)$$

Consider the running example illustrated in Figure 2. PE0 and PE1 communicate only through the data link $\vec{\ell} = (1)$. Because $|\mathbf{L}| = 1$, each tile generates one communication set $\mathbf{C}_{(t_1, t_2), (1)}$ as depicted in Figure 2(i). $\mathbf{C}_{(t_1, t_2), (1)}$ contains one element if tile (t_1, t_2) is located in the top row and two elements otherwise. In the proposed approach, the communication set will be approximated as containing two elements. Each tile (t_1, t_2) generates two nonlocal access sets, $\mathbf{N}_{(t_1, t_2), (1, 0)}$ and $\mathbf{N}_{(t_1, t_2), (1, 1)}$, one element per set, as depicted in Figure 2(j). These two nonlocal access sets are accessed in tiles $(t_1 + 1, t_2)$ and $(t_1 + 1, t_2 + 1)$, respectively. Of course, if (t_1, t_2) is a tile in the top row, $\mathbf{N}_{(t_1, t_2), (1, 1)}$ will not be accessed because tile $(t_1 + 1, t_2 + 1)$ does not exist.

Recall that $\vec{\delta}(\vec{d})$ contains all tile dependence vectors induced by the dependence vector \vec{d} . Let

$$\mathbf{D}_{\vec{\delta}} = \{\vec{d} \mid \forall \vec{d}' \in \mathbf{D} : \vec{\delta}(\vec{d}') = \vec{\delta}\}$$

Thus $\mathbf{D}_{\vec{\delta}}$ contains all dependence vectors in the program that induce the tile dependence vector $\vec{\delta}$.

The nonlocal access set $\mathbf{N}_{\vec{t}, \vec{\delta}}$ contains the data produced in tile \vec{t} and read by all possible references $A(f(\vec{i} - \vec{d}))$ in tile $\vec{t} + \vec{\delta}$, where $\vec{d} \in \mathbf{D}_{\vec{\delta}}$. So it can be expressed as follows:

$$\mathbf{N}_{\vec{t}, \vec{\delta}} = \{f(\vec{i}) \mid \vec{i} \in \mathbf{I}_{\vec{t}} \wedge (\exists \vec{d} \in \mathbf{D}_{\vec{\delta}} : (\vec{i} + \vec{d}) \in \mathbf{I}_{\vec{t} + \vec{\delta}})\}$$

which is equivalent to:

$$\mathbf{N}_{\vec{t}, \vec{\delta}} = \bigcup_{\vec{d} \in \mathbf{D}_{\vec{\delta}}} \{f(\vec{i}) \mid \vec{i} \in \mathbf{I}_{\vec{t}} \wedge (\vec{i} + \vec{d}) \in \mathbf{I}_{\vec{t} + \vec{\delta}}\}$$

By using (12), $\mathbf{C}_{\vec{t}, \vec{\ell}}$ can be expressed as:

$$\mathbf{C}_{\vec{t}, \vec{\ell}} = \bigcup_{\forall \vec{\delta} \in \Delta: \vec{\delta}[1:m] = \vec{\ell}} \left(\bigcup_{\vec{d} \in \mathbf{D}_{\vec{\delta}}} \{f(\vec{i}) \mid \vec{i} \in \mathbf{I}_{\vec{t}} \wedge (\vec{i} + \vec{d}) \in \mathbf{I}_{\vec{t} + \vec{\delta}}\} \right)$$

Based on (3), the inequalities $\vec{i} \in \mathbf{I}_t \wedge (\vec{i} + \vec{d}) \in \mathbf{I}_{t+\delta}$ can be made more explicit:

$$\mathbf{C}_{\vec{t}, \vec{\ell}} = \bigcup_{\forall \delta \in \Delta: \delta[1:m]=\vec{\ell}} \left(\bigcup_{\vec{d} \in \mathbf{D}_{\vec{\delta}}} \{f(\vec{i}) \mid \begin{aligned} &\vec{i} \in \mathbf{I} \wedge (\vec{i} + \vec{d}) \in \mathbf{I} \wedge \\ &\vec{0} \leq \vec{i} - \vec{i}_{\min} - \vec{t} \circ \vec{B} \leq \vec{B} - \vec{1} \wedge \\ &-\vec{d} \leq \vec{i} - \vec{i}_{\min} - (\vec{t} + \vec{\delta}) \circ \vec{B} \leq \vec{B} - \vec{d} - \vec{1} \} \right) \end{aligned}$$

If $\vec{i} \in \mathbf{I} \wedge (\vec{i} + \vec{d}) \in \mathbf{I}$ is ignored, this set is over-approximated only for boundary tiles. We obtain:

$$\mathbf{C}_{\vec{t}, \vec{\ell}} = \bigcup_{\forall \delta \in \Delta: \delta[1:m]=\vec{\ell}} \left(\bigcup_{\vec{d} \in \mathbf{D}_{\vec{\delta}}} \{f(\vec{i}) \mid \begin{aligned} &\vec{0} \leq \vec{i} - \vec{i}_{\min} - \vec{t} \circ \vec{B} \leq \vec{B} - \vec{1} \wedge \\ &-\vec{d} \leq \vec{i} - \vec{i}_{\min} - (\vec{t} + \vec{\delta}) \circ \vec{B} \leq \vec{B} - \vec{d} - \vec{1} \} \right) \end{aligned} \quad (13)$$

which can be simplified to:

$$\mathbf{C}_{\vec{t}, \vec{\ell}} = \bigcup_{\vec{d} \in \mathbf{D}_{\vec{\delta}}} \{f(\vec{i}) \mid \begin{aligned} &\vec{0} \leq (\vec{i} - \vec{i}_{\min} - \vec{t} \circ \vec{B}) \leq \vec{B} - \vec{1} \wedge \\ &-\vec{d}[1:m] \leq (\vec{i} - \vec{i}_{\min})[1:m] - (\vec{t}[1:m] + \vec{\ell}) \circ \vec{B}[1:m] \leq (\vec{B} - \vec{d})[1:m] - \vec{1} \} \end{aligned} \quad (14)$$

where $\vec{\delta}' = \begin{pmatrix} \vec{\ell} \\ \vec{0} \end{pmatrix}$.

Let $\vec{d}_{\vec{\ell}, \max}$ and $\vec{d}_{\vec{\ell}, \min}$ be defined such that their k -th entries are:

$$\begin{aligned} d_{\vec{\ell}, \max, k} &= \max\{d_k \mid \vec{d} \in \mathbf{D}_{\vec{\delta}'}\} \\ d_{\vec{\ell}, \min, k} &= \min\{d_k \mid \vec{d} \in \mathbf{D}_{\vec{\delta}'}\} \end{aligned} \quad (15)$$

The communication set $\mathbf{C}_{\vec{t}, \vec{\ell}}$ is further over-approximated as the smallest enclosing rectangle:

$$\mathbf{C}_{\vec{t}, \vec{\ell}} = \{f(\vec{i}) \mid \begin{aligned} &\vec{0} \leq (\vec{i} - \vec{i}_{\min} - \vec{t} \circ \vec{B}) \leq \vec{B} - \vec{1} \wedge \\ &-\vec{d}_{\vec{\ell}, \max}[1:m] \leq (\vec{i} - \vec{i}_{\min})[1:m] - (\vec{t}[1:m] + \vec{\ell}) \circ \vec{B}[1:m] \leq (\vec{B} - \vec{d}_{\vec{\ell}, \min})[1:m] - \vec{1} \} \end{aligned} \quad (16)$$

Here, \vec{B} , $\vec{d}_{\vec{\ell}, \max}$ and $\vec{d}_{\vec{\ell}, \min}$ are all compile-time constants, \vec{i}_{\min} is a symbolic constant as a function of problem size parameters and \vec{t} consists of the n tile indices. Thus, the communication set is a convex polyhedron in terms of the n loop indices i_1, \dots, i_n . Therefore, the compiler can use these inequalities to generate a set of perfectly nested loops to pack and unpack messages.

The communication set approximated this way is expected to be exact for most practical cases. For example, if $\forall \vec{d} \in \mathbf{D}_{\vec{\delta}'} : \vec{d}[1:m] \in \{0, 1\}^m$, then (16) defines exactly the same set as (14). In this case, the communication set $\mathbf{C}_{\vec{t}, \vec{\ell}}$ is exact unless \vec{t} is close to the iteration space boundaries.

To find out the inequalities that specify the communication set $\mathbf{C}_{(t_1, t_2), (1)}$ for the running example depicted in Figure 2(i), we simply substitute all known parameters into (16). By using $\vec{B} = (2, 2)$, $\vec{\ell} = (1)$, $\vec{d}_{\vec{\ell}, \max} = \vec{d}_{\vec{\ell}, \min} = (1, 1)$ and $\vec{i}_{\min} = (1, 1)$, we obtain (with simplifications):

$$\mathbf{C}_{(t_1, t_2), (1)} = \{(i, 2j) \mid i = 2t_1 + 2 \wedge 2t_2 + 1 \leq j \leq 2t_2 + 2\}$$

This communication set is exact for all tiles except those on the top of the iteration space (Figure 2(i)). For a tile on the top, one element is transferred redundantly.

```

1  for  $\vec{\ell} \in \mathbf{L}$ 
2    if ( $\forall \vec{s} \in \text{succ}(\vec{t}, \vec{\ell}) : \text{valid}(\vec{s}) = \text{false}$ ) continue;
3    len := 0;
4    for  $\vec{i} \in \mathbf{C}_{\vec{t}, \vec{\ell}}$  in lexicographic order  $\prec$ 
5      buf[len++] :=  $A(f(\vec{i}))$ ;
6    send( $((\vec{p} + \vec{\ell}) \bmod \vec{P}, \text{buf}, \text{len})$ );

```

(a) Send code for “Pack and Send Messages between Tiles”

```

1  for  $\vec{\ell} \in \mathbf{L}$ 
2    for  $\vec{s} \in \text{pred}(\vec{t}, \vec{\ell})$  in lexicographic order  $\prec$ 
3      if  $\text{valid}(\vec{s}) = \text{false}$  continue;
4      if  $\vec{t} = \text{minsucc}(\vec{s}, \vec{\ell})$  then
5        recv( $((\vec{p} - \vec{\ell}) \bmod \vec{P}, \text{buf}, \text{sizeof}(\text{buf}))$ );
6        len := 0;
7        for  $\vec{i} \in \mathbf{C}_{\vec{s}, \vec{\ell}}$  in lexicographic order  $\prec$ 
8           $A(f(\vec{i})) := \text{buf}[\text{len}++]$ ;

```

(b) Receive code for “Receive and Unpack Messages between Tiles”

Figure 5: Two message-passing code sections for tiles.

5.2 Message-Passing Code

A tile \vec{t} is said to be *valid* if it is enumerated by the tile loops, i.e., if it is in the effective tile space \mathbf{T}' . So an empty tile is also regarded as being valid if it is enumerated by the tile loops. It is possible to solve an integer programming problem to check at run-time if every tile being enumerated is empty or not [28]. But this will be too expensive and unnecessary because the tile loops are expected to be exact for real programs, and if not, the number of empty tiles should be very small. On the other hand, it is efficient to check at run-time whether a tile \vec{t} is valid or not by simply evaluating the conditional expression $\vec{t} \in \mathbf{T}'$.

Let $\text{valid}(\vec{t})$ be a boolean function that returns true if \vec{t} is a valid tile and false otherwise.

During the execution of our SPMD code, a processor does not distinguish the empty tiles from those that are not. However, due to the cyclic computation and data distribution, a processor has to verify the validity of some tiles at run-time in order to send and receive messages correctly.

Let $\text{minsucc}(\vec{t}, \vec{\ell})$ be the lexicographically minimum valid tile in the successor set $\text{succ}(\vec{t}, \vec{\ell})$:

$$\text{minsucc}(\vec{t}, \vec{\ell}) = \min_{\prec} \{ \vec{t} \mid (\forall \vec{t} \in \text{succ}(\vec{t}, \vec{\ell}) : \text{valid}(\vec{t}) = \text{true}) \} \quad (17)$$

Figure 5 gives both the send and receive code for implementing the middle two message-passing code sections from Figure 4. In the send code, each tile \vec{t} executed at a processor \vec{p} generates exactly $|\mathbf{L}|$ communication sets (i.e., messages) $\mathbf{C}_{\vec{t}, \vec{\ell}}$, one for each neighbouring processor $(\vec{p} + \vec{\ell}) \bmod \vec{P}$ such that $\vec{\ell} \in \mathbf{L}$. This message is delivered as long as there is one valid successor tile in $\text{succ}(\vec{t}, \vec{\ell})$. In the receive code, a processor \vec{p} must ensure that all messages generated from the predecessor tiles

in $\text{pred}(\vec{t}, \vec{\ell})$ for every $\vec{\ell} \in \mathbf{L}$ have been received before it can execute tile \vec{t} . Let $\vec{s} \prec \vec{t}$ be a tile that is also allocated to the processor \vec{p} and that depends also on some of the predecessor tiles in $\text{pred}(\vec{t}, \vec{\ell})$. Thus, the messages generated in the predecessor tiles of $\text{pred}(\vec{t}, \vec{\ell})$ that \vec{s} depends must be received even before \vec{s} is executed. So the key in the construction of the receive code is to decide when a message should be received. The answer is that a message should be received just before the first tile that depends on it is executed. This first tile is identified by $\text{minsucc}(\vec{t}, \vec{\ell})$ in line 4 of the receive code shown in Figure 5(b). In line 2 of the receive code, the for loop processes all the predecessor tiles in lexicographic order and receives the communication sets generated in these tiles if tile \vec{t} is the first consumer of these messages.

The receive code is constructed to work for any arbitrary iteration space and can be simplified for commonly occurring iteration spaces such as rectangular and triangular iteration spaces. Section 5.3 demonstrates how this can be achieved when the iteration space is rectangular.

THEOREM 2 *The communication code in Figure 5 ensures that all cross-processor tile dependences are satisfied before a tile is executed.*

Proof. Suppose processor \vec{p} is ready to execute tile \vec{t} . The tile \vec{t} consumes the data produced at the neighbouring processor $(\vec{p} - \vec{\ell}) \bmod \vec{P}$ from all valid tiles \vec{s} in the predecessor set $\text{pred}(\vec{t}, \vec{\ell})$. Let the tiles $\vec{s}_1, \dots, \vec{s}_r$ in $\text{pred}(\vec{t}, \vec{\ell})$ be ordered such that $\vec{s}_1 \prec \dots \prec \vec{s}_r$. These tiles are executed in lexicographic order. Therefore, the communication sets $\mathbf{C}_{\vec{s}_1, \vec{\ell}}, \dots, \mathbf{C}_{\vec{s}_r, \vec{\ell}}$ generated in these tiles are sent to the processor \vec{p} in that order; so they should be received in the same order (line 2). Of course, if a tile in $\text{pred}(\vec{t}, \vec{\ell})$ is invalid – which can happen when \vec{t} is a close to the boundaries of the effective tile space, then there is nothing to receive (line 3). Consider the k -th such communication set $\mathbf{C}_{\vec{s}_k, \vec{\ell}}$. By the definition of (16), it carries the data to be consumed by all valid tiles in the successor $\text{succ}(\vec{s}_k, \vec{\ell})$ that are executed at the processor \vec{p} , and tile \vec{t} is among these consumer tiles. Clearly, the processor must receive this k -th communication set when $\vec{t} = \text{minsucc}(\vec{t}, \vec{\ell})$ (line 4), i.e., when \vec{t} is the first tile that accesses the data in the message $\mathbf{C}_{\vec{s}_k, \vec{\ell}}$. If $\text{minsucc}(\vec{s}_k, \vec{\ell}) \prec \vec{t}$ instead, then the message $\mathbf{C}_{\vec{s}_k, \vec{\ell}}$ was received before tile $\text{minsucc}(\vec{s}_k, \vec{\ell})$ is executed. Therefore, all cross-processor dependences will be first satisfied before a tile can be executed. ■

In both the send and receive code, the test $\text{valid}(\vec{s})$, can be replaced with $\vec{t} \in \mathbf{T}' \Rightarrow \vec{s} \in \mathbf{T}'$ and simplified symbolically using the techniques promoted in the Omega Calculator [28]. In practice, the predicate we use in line 2 of the send code is $\vec{t} \in \mathbf{T}' \Rightarrow (\forall \vec{s} \in \text{succ}(\vec{t}, \vec{\ell}) : \vec{s} \notin \mathbf{T}')$ after being symbolically simplified. In line 4 of the receive code, $\text{minsucc}(\vec{s}, \vec{\ell})$ for all predecessor tiles \vec{s} of tile \vec{t} are found efficiently in one-pass as follows. Let $\vec{s}_{\min} = \min_{\prec} \text{pred}(\vec{t}, \vec{\ell})$. We have $\text{succ}(\vec{s}_{\min}, \vec{\ell}) \supseteq \{\vec{t}' \preceq \vec{t} \mid \vec{t}' \in \text{succ}(\vec{s}, \vec{\ell})\}$ for every \vec{s} in $\text{pred}(\vec{t}, \vec{\ell})$. The successor set $\text{succ}(\vec{s}_{\min}, \vec{\ell})$ contains at most 2^{n-m} elements. For instance, $\text{succ}(\vec{s}_{\min}, \vec{\ell})$ contains at most two elements when a 3D iteration space is distributed to a 2D processor space, where $n = 3$ and $m = 2$). The compiler can generate code to check if each tile in $\text{succ}(\vec{s}_{\min}, \vec{\ell})$ is valid or not, and find $\text{minsucc}(\vec{s}, \vec{\ell})$ for all $\vec{s} \in \text{pred}(\vec{t}, \vec{\ell})$ in one pass. Our experimental results indicate that the time taken by this part of the code is negligible.

```

1  for  $\vec{\ell} \in \mathbf{L}$ 
2    if  $(t_1 + \ell_1 > U_1^t \vee \dots \vee t_m + \ell_m > U_m^t)$  continue;
3    len := 0;
4    for  $\vec{i} \in \mathbf{C}_{\vec{t}, \vec{\ell}}$  in lexicographic order  $\prec$ 
5      buf[len++] :=  $A(f(\vec{i}))$ ;
6    send( $(\vec{p} + \vec{\ell}) \bmod \vec{P}$ , buf, len);

```

(a) Simplified send code

```

1  for  $\vec{\ell} \in \mathbf{L}$ 
2    if  $(L_1^t > t_1 - \ell_1 \vee \dots \vee L_m^t > t_m - \ell_m)$  continue;
3    recv( $(\vec{p} - \vec{\ell}) \bmod \vec{P}$ , buf, sizeof(buf));
4    len := 0;
5    for  $\vec{i} \in \mathbf{C}_{\vec{t} - (\ell_1, \dots, \ell_m, 0, \dots, 0), \vec{\ell}}$  in lexicographic order  $\prec$ 
6       $A(f(\vec{i})) := \text{buf}[\text{len}++]$ ;

```

(b) Simplified receive code

Figure 6: Simplified message-passing code for rectangular iteration spaces.

5.3 Simplified Message-Passing Code for Rectangular Iteration Spaces

If the iteration space \mathbf{I} is rectangular, the effective tile space \mathbf{T}' will be rectangular as well. Then all successor tiles in $\text{succ}(\vec{t}, \vec{\ell})$ are either simultaneously valid or not. These successor tiles are all valid if and only if $(\forall k : t_k + \ell_k \leq U_k^t)$. This leads to the simplified send code given in Figure 6(a). Similarly, all tiles in the predecessor set $\text{pred}(\vec{t}, \vec{\ell})$ are either simultaneously valid or not. These predecessor tiles are all valid if and only if $(\forall k : L_k^t \leq t_k - \ell_k)$. For every predecessor tile \vec{s} in $\text{pred}(\vec{t}, \vec{\ell})$, all of the successor tiles in $\text{succ}(\vec{s}, \vec{\ell})$ must be valid. A further application of the following theorem leads directly to the simplified receive code shown in Figure 6(b).

THEOREM 3 *Let \vec{t} be a valid tile. Let $\vec{s} \in \text{pred}(\vec{t}, \vec{\ell})$ be a valid tile. If all tiles in $\text{succ}(\vec{s}, \vec{\ell})$ are valid, then $\text{minsucc}(\vec{s}, \vec{\ell}) = \vec{t}$ if and only if $\vec{s} = \vec{t} - (\ell_1, \dots, \ell_m, 0, \dots, 0)$.*

Proof. By construction, \vec{s} must have the form: $\vec{s} = \vec{t} + (-\ell_1, \dots, -\ell_m, -\delta_{m+1}, \dots, -\delta_n)$, where $(\delta_{m+1}, \dots, \delta_n) \in \{0, 1\}^{n-m}$. If all tiles in $\text{succ}(\vec{s}, \vec{\ell})$ are valid, then $\text{succ}(\vec{s}, \vec{\ell})$ must contain $\vec{s} + (\vec{\ell}_1, \dots, \vec{\ell}_m, 0, \dots, 0)$. This implies that $\text{minsucc}(\vec{s}, \vec{\ell}) = \vec{s} + (\ell_1, \dots, \ell_m, 0, \dots, 0)$. Since $\vec{s} = \vec{t} + (-\ell_1, \dots, -\ell_m, -\delta_{m+1}, \dots, -\delta_n)$, we have $\text{minsucc}(\vec{s}, \vec{\ell}) = \vec{t} + (0, \dots, 0, -\delta_{m+1}, \dots, -\delta_n)$. Hence, if $\text{minsucc}(\vec{s}, \vec{\ell}) = \vec{t}$, implying that $\delta_{m+1} = \dots = \delta_n = 0$, then $\vec{s} = \vec{t} - (\ell_1, \dots, \ell_m, 0, \dots, 0)$, and conversely. ■

By inserting the the communication code into the program given in (9), we obtain the SPMD program for our running program given in Figure 7, where the I/O code is not shown.

```

for( $t_1 = p_1; t_1 \leq 4; t_1 += 2$ )
  for( $t_2 = 0; t_2 \leq 1; t_2 ++$ )
    /* Receive message  $C_{(t_1-1, t_2), (1)}$  */
    if  $t_1 < 1$  continue;
    recv( $(p_1 - 1) \bmod 2$ , buf, sizeof(buf));
    len := 0;
    for( $i = 2t_1; i \leq 2t_1; i ++$ )
      for( $j = 2t_2 + 1; j \leq 2t_2 + 2; j ++$ )
         $A(i, 2j) = \text{buf}[\text{len}++]$ ;
    for( $i = \max(1, 2t_1 + 1); i \leq \min(9, 2t_1 + 2); i ++$ )
      for( $j = \max(1, 2t_2 + 1); j \leq \min(4, 2t_2 + 2); j ++$ )
         $A(i, 2j) = A(i, 2j - 2) + A(i - 1, 2j - 2)$ 
    /* Send message  $C_{(t_1, t_2), (1)}$  */
    if  $t_1 > 3$  continue;
    len := 0;
    for( $i = 2t_1 + 2; i \leq 2t_1 + 2; i ++$ )
      for( $j = 2t_2 + 1; j \leq 2t_2 + 2; j ++$ )
        buf[ $\text{len}++$ ] =  $A(i, 2j)$ ;
    send( $(p_1 + 1) \bmod 2$ , buf, len);

```

Figure 7: SPMD program with communication code for Example 1.

6 Memory Management

Because we create SPMD programs, all processors must possess the same array declarations. Therefore, each processor will execute in its own local address space. For regularly structured computations, we use a storage scheme to store both the local and nonlocal data of a distributed array in the same local array. To hold the nonlocal data, we use an extension of *overlap areas* [15]: the nonlocal data are stored according to the loop indices at which they are computed rather than their array indices. This storage scheme leads to high locality of references for the computations within tiles regardless of any sparse array references that may appear in the original program.

Let us use the program given in Figure 7 to illustrate our storage scheme. Figure 8(a) depicts the portion of the data space of A , where the elements accessed at PE1 are indicated with their array indices. The superscripts denote the tile indices at which the local elements are computed (see Figure 2(b)). The shaded entries are nonlocal elements: those in light gray are the read-only data received from the host and the others are the nonlocally computed data received from PE0. In Figure 8(b), the elements distributed to PE1 are compressed to yield a compact local array \mathcal{A} . The nonlocal elements are stored close to where they are used. The arrows indicate the two dependence vectors $(0, 1)$ and $(1, 1)$ between the elements in the local array.

In general, the local memory space in \mathcal{A} is organised as follows. The data computed within a tile are stored in an array section of size $B_1 \times \dots \times B_n$. The nonlocal data (including the read-only data) accessed in a distributed dimension are stored in the overlap area of the array section on the side towards the negative direction of the axis; the other side is not expanded because only the flow dependences are considered. To store the read-only data initially received from the host in a non-distributed dimension, each local array is also expanded at the beginning in that dimension.

		$3, 8^1$	$4, 8^1$			$7, 8^1$	$8, 8^1$	
	$2, 6$	$3, 6^1$	$4, 6^1$		$6, 6$	$7, 6^1$	$8, 6^1$	
	$2, 4$	$3, 4^0$	$4, 4^0$		$6, 4$	$7, 4^0$	$8, 4^0$	
	$2, 2$	$3, 2^0$	$4, 2^0$		$6, 2$	$7, 2^0$	$8, 2^0$	
	$2, 0$	$3, 0$	$4, 0$		$6, 0$	$7, 0$	$8, 0$	

(a) The elements of A accessed at PE1

		$A(3, 8)$	$A(4, 8)$		$A(7, 8)$	$A(8, 8)$	
$A(2, 6)$	$A(3, 6)$	$A(4, 6)$	$A(6, 6)$	$A(7, 6)$	$A(8, 6)$		
$A(2, 4)$	$A(3, 4)$	$A(4, 4)$	$A(6, 4)$	$A(7, 4)$	$A(8, 4)$		
$A(2, 2)$	$A(3, 2)$	$A(4, 2)$	$A(6, 2)$	$A(7, 2)$	$A(8, 2)$		
$A(2, 0)$	$A(3, 0)$	$A(4, 0)$	$A(6, 0)$	$A(7, 0)$	$A(8, 0)$		

(b) The elements of A stored in the local array \mathcal{A}

Figure 8: Local memory allocation for PE1 for the program in Figure 7.

The local data space $\mathbf{I}_{\vec{p}}$ allocated to a processor is non-contiguous because the data distribution is cyclic and can be sparse because array references can be sparse. For instance, the array reference function in Example 1, as illustrated in Figure 8, is sparse. In our storage scheme, both kinds of sparsity are removed.

Our storage scheme can be summarised in one sentence: *Any two dependent elements $A(f(\vec{i}))$ and $A(f(\vec{i} - \vec{d}))$ are mapped to the local array elements $\mathcal{A}(\vec{a})$ and $\mathcal{A}(\vec{a} - \vec{d})$, respectively, where \vec{a} is an array index vector for the local array \mathcal{A} .* Regrettably, the various address translation functions required for achieving this objective can be quite involved both technically and notationally.

In Section 6.1, we present a memory compression technique to determine the minimal amount of local memory allocated for a distributed array. Section 6.2 discusses the conversion between global and local loop indices. In Section 6.3, we describe the required translations between global and local array indices to find out the addresses to store (a) locally computed data, (b) the read-only data received from the host, and (c) the nonlocal data received from neighbouring processors.

Let \vec{d}_{\max} be defined such that its k -th entry $d_{\max, k}$ is the maximum of the k -th entries of all dependence vectors in \mathbf{D} : $d_{\max, k} = \max\{d_k \mid \vec{d} \in \mathbf{D}\}$.

6.1 Local Array Allocation

For each array A in the program, all processors use the same local array $\mathcal{A}[0 : u_1, \dots, 0 : u_n]$ to store both the local and nonlocal elements.

Let \vec{i}_{\max} be the smallest integer vector such that $\vec{i}_{\max} \geq \vec{i}$ for all iterations \vec{i} in the iteration space. That is, $i_{\max, k} = \max(i_k \mid (i_1, \dots, i_n) \in \mathbf{I})$. Just like its counterpart \vec{i}_{\min} , \vec{i}_{\max} can be found using Fourier-Motzkin elimination [31, p. 49] or the PIP system [12].

In the k -th dimension, the maximal number of tiles is $\lceil \frac{i_{\max, k} - i_{\min, k+1}}{B_k} \rceil$. Two cases are considered. If k is a distributed dimension, where $1 \leq k \leq m$, then the maximal number of these tiles allocated to any processor is at most $\lceil \frac{i_{\max, k} - i_{\min, k+1}}{B_k P_k} \rceil$. To accommodate both the local data and the nonlocal data (including the read-only data), the size of the k -th distributed dimension required is at most

$$(B_k + d_{\max,k}) \lceil \frac{i_{\max,k} - i_{\min,k} + 1}{B_k P_k} \rceil.$$

If the k -th dimension is not distributed, where $m < k \leq n$, the size of \mathcal{A} along that dimension is $B_k \lceil \frac{i_{\max,k} - i_{\min,k} + 1}{B_k} \rceil + d_{\max,k}$, where $d_{\max,k}$ is needed to store the read-only data from the host.

Let $\vec{B}' = (B_1 + d_{\max,1}, \dots, B_m + d_{\max,m}, B_{m+1}, \dots, B_n)$. The upper bounds in $\mathcal{A}[0 : u_1, \dots, 0 : u_n]$ are defined as follows:

$$u_k = \begin{cases} \text{if } 1 \leq k \leq m & \rightarrow B'_k \lceil \frac{i_{\max,k} - i_{\min,k} + 1}{B_k P_k} \rceil - 1 \\ \text{[] else} & \rightarrow B'_k \lceil \frac{i_{\max,k} - i_{\min,k} + 1}{B_k} \rceil + d_{\max,k} - 1 \\ \text{fi} & \end{cases} \quad (18)$$

In the case of the SPMD program given in Figure 7, we have $\vec{i}_{\min} = (1, 1)$, $\vec{i}_{\max} = (9, 4)$, $\vec{d}_{\max} = (1, 1)$, $(B_1, B_2) = (2, 2)$, $(B'_1, B'_2) = (3, 2)$. So the local array declarations used is $\mathcal{A}[0 : 8, 0 : 4]$.

6.2 Global v.s. Local Loop indices

A function that translates between global loop indices to local loop indices is given. This function will be used to define various address translation functions to map both the local and nonlocal elements of an array to the local memory of a processor.

For notational convenience, let $\vec{P}' = (P_1, \dots, P_m, 1, \dots, 1) \in \mathbb{Z}^n$.

The global loop indices are mapped to the local loop indices as follows:

$$\theta : \mathbf{I} \mapsto \mathbb{Z}^n, \theta(\vec{i}) = \vec{B}' \circ \lfloor \frac{\vec{i} - \vec{i}_{\min}}{\vec{B} \circ \vec{P}'} \rfloor + (\vec{i} - \vec{i}_{\min}) \bmod \vec{B} + \vec{d}_{\max} \quad (19)$$

The conversion from global to local loop indices is independent of the processor identifier. Let $\mathbf{I}_\ell = \{\theta(\vec{i}) \mid \vec{i} \in \mathbf{I}\}$. The inverse of this function is found to be:

$$\theta_{\vec{p}}^{-1} : \mathbf{I}_\ell \mapsto \mathbf{I}, \theta_{\vec{p}}^{-1}(\vec{i}') = \vec{B} \circ (\vec{P}' \circ \lfloor \frac{\vec{i}' - \vec{d}_{\max}}{\vec{B}'} \rfloor + \vec{p}) + (\vec{i}' - \vec{d}_{\max}) \bmod \vec{B} + \vec{i}_{\min} \quad (20)$$

The inverse is dependent on the processor identifier \vec{p} .

The division and modulo operations are expensive, especially if they are performed frequently inside the innermost loop of the SPMD code. When generating SPMD code, the following fact will be exploited. Let $\vec{i}' = \theta(\vec{i})$. By noting that $\vec{t} = \lfloor \frac{\vec{i}' - \vec{i}_{\min}}{\vec{B}} \rfloor$ and $(\vec{i}' - \vec{i}_{\min}) \bmod \vec{B} = \vec{i}' - \vec{B} \circ \vec{t} - \vec{i}_{\min}$, we can simplify the function θ by avoiding the modulo operation:

$$\theta : \mathbf{I} \mapsto \mathbb{Z}^n, \theta(\vec{i}) = \vec{B}' \lfloor \frac{\vec{i}'}{\vec{P}'} \rfloor + \vec{i}' - \vec{B} \circ \vec{t} - \vec{i}_{\min} + \vec{d}_{\max} \quad (21)$$

The division and modulo operations are unnecessary for a non-distributed dimension k , where $m < k \leq n$. Because $\vec{B}' = (B_1 + d_{\max,1}, \dots, B_m + d_{\max,m}, B_{m+1}, \dots, B_n)$ and $\vec{P}' = (P_1, \dots, P_m, 1, \dots, 1)$, the k -th component of the function θ can be simplified to:

$$\theta(\vec{i})_k = i_k - i_{\min,k} + d_{\max,k} \quad (22)$$

and similarly, the k -th component of the function $\theta_{\vec{p}}^{-1}$ can be simplified to:

$$\theta_{\vec{p}}^{-1}(\vec{i}')_k = i'_k - d_{\max,k} + i_{\min,k}$$

6.3 Global v.s. Local Array indices

Given an array A , this section provides the formulas that a processor \vec{q} can use to find out the local addresses to store the three types of data elements for A : (a) the local data $\mathbf{O}_{\vec{q}}$ defined in (10); (b) the read-only data $\mathbf{O}_{\text{host},\vec{q}}$ defined in (11), which are received initially from the host; and (c) the nonlocal data $\mathbf{C}_{\vec{i},\vec{\ell}}$ defined in (16), which are received from the neighbouring processor $(\vec{q} - \vec{\ell}) \bmod \vec{P}$.

As mentioned earlier, all address translation functions are defined so that two dependent elements $A(f(\vec{i}))$ and $A(f(\vec{i} - \vec{d}))$ are mapped to the local array elements $\mathcal{A}(\vec{a})$ and $\mathcal{A}(\vec{a} - \vec{d})$, respectively.

6.3.1 Mapping Locally Computed Data

A locally computed element $A(\vec{a})$ in $\mathbf{O}_{\vec{q}}$ is mapped to $\mathcal{A}(\lambda(\vec{a}))$ as follows:

$$\lambda(\vec{a}) = \theta(f^{-1}(\vec{a})) \quad (23)$$

The inverse of this function given below depends on the processor identifier \vec{q} :

$$\lambda_{\vec{q}}^{-1}(\vec{a}') = f(\theta_{\vec{q}}^{-1}(\vec{a}'))$$

The effect of f on the sparsity of array accesses is eliminated since for every write reference $A(\vec{a})$, there must exist an iteration \vec{i} such that $\vec{a} = f(\vec{i})$. This means that $\lambda(\vec{a}) = \theta(f^{-1}(\vec{a})) = \theta(f^{-1}(f(\vec{i}))) = \theta(\vec{i})$. So it is the iteration vector \vec{i} rather than the array index vector \vec{a} that determines the actual mapping of $A(\vec{a})$ to the local processor memory. Once the effect of f on array references is eliminated, the function θ has been designed to compress the loop indices resulting from a cyclical computation distribution into contiguous local array indices.

This function tells where the results of the computation are. Thus, the compiler can use this function to generate the bottom code section in Figure 4 to send the results to the host.

6.3.2 Mapping the Read-Only Data

Each read-only element $A(\vec{a})$ in $\mathbf{O}_{\text{host},\vec{q}}$ is mapped to $\mathcal{A}(\eta(\vec{a}))$ according to the function:

$$\eta(\vec{a}) = \theta(f^{-1}(\vec{a}) + \vec{d}) - \vec{d}, \text{ where } \vec{d} \in \mathbf{D} \text{ such that } f^{-1}(\vec{a}) + \vec{d} \in \mathbf{I}_{\vec{q}} \quad (24)$$

Here, $f^{-1}(\vec{a}) + \vec{d}$ gives the iteration at which $A(\vec{a})$ is accessed.

This function is used in the top code section in Figure 4 to store the read-only data appropriately.

6.3.3 Mapping Nonlocally Computed Data

Suppose that processor \vec{q} has received the nonlocal data elements in the communication set $\mathbf{C}_{\vec{i},\vec{\ell}}$ from the processor $(\vec{q} - \vec{\ell}) \bmod \vec{P}$. Each nonlocal element $A(\vec{a})$ in $\mathbf{C}_{\vec{i},\vec{\ell}}$ can be mapped to $\mathcal{A}(\eta(\vec{a}))$ using exactly the same function η for mapping the read-only data:

$$\eta(\vec{a}) = \theta(f^{-1}(\vec{a}) + \vec{d}) - \vec{d}, \text{ where } \vec{d} \in \mathbf{D} \text{ such that } f^{-1}(\vec{a}) + \vec{d} \in \mathbf{I}_{\vec{q}} \quad (25)$$

Since the iteration $f^{-1}(\vec{a})$ is computed by the processor $(\vec{q} - \vec{\ell}) \bmod \vec{P}$, $f^{-1}(\vec{a}) + \vec{d}$ must be computed at the receiving processor \vec{q} .

In practice, however, the nonlocally computed data are not mapped this way to the local memory of the receiving processor \vec{q} . Because each processor executes in its own local address space, the nonlocal data contained in a message is expressed in the local address space of the sending processor $(\vec{q} - \vec{\ell}) \bmod \vec{P}$. These nonlocal data must be mapped to the local address space of the receiving processor \vec{q} .

By applying the function λ to the communication set $\mathcal{C}_{\vec{i}, \vec{\ell}}$ in the global address space, we obtain the following communication set in the local address space of the sending processor $\vec{p} = \vec{q} - \vec{\ell} \bmod \vec{P}$:

$$\mathcal{C}_{\vec{i}, \vec{\ell}} = \{ \vec{i}' \mid \vec{0} \leq \vec{i}' - \vec{B}' \lfloor \frac{\vec{i}}{\vec{P}} \rfloor - \vec{d}_{\max} \leq \vec{B} - \vec{1} \\ - \vec{d}_{\vec{\ell}, \max}[1 : m] \leq (\vec{i}' - \vec{B}' \lfloor \frac{\vec{i}}{\vec{P}} \rfloor - \vec{d}_{\max})[1 : m] - \vec{\ell} \circ \vec{B}[1 : m] \leq \vec{B}[1 : m] - \vec{d}_{\vec{\ell}, \min}[1 : m] - \vec{1} \} \quad (26)$$

Each communication set now represents a regular array section of the local array \mathcal{A} , yielding more efficient packing and unpacking code. When some boundary tiles are executed, some extra elements of \mathcal{A} transferred may not be associated with any elements in the distributed array A . But these elements will never be accessed.

When the processor \vec{q} receives the communication set $\mathcal{C}_{\vec{i}, \vec{\ell}}$ from $\vec{p} = (\vec{q} - \vec{\ell}) \bmod \vec{P}$, it uses the following function to map a data element $\mathcal{A}(\vec{a}')$ in $\mathcal{C}_{\vec{i}, \vec{\ell}}$ to $\mathcal{A}(\zeta(\vec{a}'))$ in its own local memory:

$$\zeta(a'_1, \dots, a'_n) = (a''_1, \dots, a''_m, a'_{m+1}, \dots, a'_n)$$

where the first m entries a''_k are defined as follows:

$$a''_k = \begin{cases} \text{if } \ell_k = 0 & \rightarrow a'_k \\ \square \text{ else} & \rightarrow \begin{cases} \text{if } q_k = 0 & \rightarrow a'_k + d_{\max, k} \\ \square \text{ else} & \rightarrow a'_k - B_k \\ \text{fi} \end{cases} \\ \text{fi} \end{cases} \quad (27)$$

LEMMA 1 *Both η and ζ map a nonlocal element received to the same local memory location of a receiving processor.*

Proof. Assume that $A(f(\vec{i}))$ computed at the processor \vec{q} accesses the non-local element $A(f(\vec{i} - \vec{d}))$ computed at the processor $(\vec{q} - \vec{\ell}) \bmod \vec{P} = \vec{p}$. According to the function η in (25), $A(f(\vec{i} - \vec{d}))$ is mapped to $\mathcal{A}(\theta(\vec{i}) - \vec{d})$ in the local address space of the receiving processor \vec{q} . We show that the function ζ in (27) will also map $A(f(\vec{i} - \vec{d}))$ to $\mathcal{A}(\theta(\vec{i}) - \vec{d})$.

Because $A(f(\vec{i} - \vec{d}))$ is a locally computed element at the sending processor $\vec{p} = (\vec{q} - \vec{\ell}) \bmod \vec{P}$, $A(f(\vec{i} - \vec{d}))$ is allocated to $\mathcal{A}(\theta(\vec{i} - \vec{d})) = \mathcal{A}(\vec{a}')$ in the local address space of \vec{p} according to the function λ in (23). The element $\mathcal{A}(\vec{a}')$ is contained in the message $\mathcal{C}_{\vec{i}, \vec{\ell}}$ sent from \vec{p} to \vec{q} . On arriving at the receiving processor \vec{q} , $\mathcal{A}(\vec{a}')$ is mapped to $\mathcal{A}(\vec{a}'')$ in the local address space of \vec{p} according to the function ζ in (27).

The rest of the proof is to establish that $\vec{a}'' = \theta(\vec{i}) - \vec{d}$, i.e., $\forall k : a''_k = \theta(\vec{i})_k - d_k$. Note that $\mathcal{A}(\vec{a}') = \mathcal{A}(\theta(\vec{i} - \vec{d}))$. Consider the k -th component of $\vec{a}' = \theta(\vec{i} - \vec{d})$. If k is a non-distributed dimension, where $m < k \leq n$, then $a''_k = a'_k = \theta(\vec{i} - \vec{d})_k = \theta(\vec{i})_k - d_k$ holds due to (22). Let us

assume that k is a distributed dimension, i.e., $1 \leq k \leq m$. Since $\vec{i} - \vec{d} \in \mathbf{I}_{\vec{t}}$, the iteration $\vec{i} - \vec{d}$ is contained in tile \vec{t} . From (21), we obtain:

$$a'_k = \theta(\vec{i} - \vec{d})_k = (B_k + d_{\max,k}) \lfloor \frac{t_k}{B_k P_k} \rfloor - i_k - d_k - B_k t_k - i_{\min,k} + d_{\max,k}$$

Let $\vec{\delta} \in \Delta$ such that $\vec{\delta}[1 : m] = \vec{\ell}$. Then $\vec{i} \in \mathbf{I}_{\vec{i} + \vec{\delta}}$. From (21), we obtain:

$$\theta(\vec{i})_k = (B_k + d_{\max,k}) \lfloor \frac{t_k + \ell_k}{B_k P_k} \rfloor - i_k - B_k(t_k + \ell_k) - i_{\min,k} + d_{\max,k}$$

If $\ell_k = 0$, $a''_k = a'_k = \theta(\vec{i})_k - d_k$ holds trivially. If $\ell_k = 1$, then two cases are considered. If $q_k = 0$, we have $(B_k + d_{\max,k}) \lfloor \frac{t_k + \ell_k}{B_k P_k} \rfloor = (B_k + d_{\max,k}) (\lfloor \frac{t_k}{B_k P_k} \rfloor + 1)$. Clearly, $a''_k = a'_k + d_{\max,k} = \theta(\vec{i})_k - d_k$ holds. If $q_k \neq 0$, we have $(B_k + d_{\max,k}) \lfloor \frac{t_k + \ell_k}{B_k P_k} \rfloor = (B_k + d_{\max,k}) \lfloor \frac{t_k}{B_k P_k} \rfloor$. Again $a''_k = a'_k - B_k = \theta(\vec{i})_k - d_k$ holds. ■

Based on this lemma, we show that the address translation functions introduced above implement the scheme as illustrated in Figure 8.

THEOREM 4 *In local address space, the loop body $A(f(\vec{i})) = F(A(f(\vec{i} - \vec{d}_1)), \dots, A(f(\vec{i} - \vec{d}_r)))$ given in Figure 4 becomes $\mathcal{A}(\theta(\vec{i})) = F(\mathcal{A}(\theta(\vec{i}) - \vec{d}_1), \dots, \mathcal{A}(\theta(\vec{i}) - \vec{d}_r))$.*

Proof. The write reference $A(f(\vec{i}))$ is mapped to $\mathcal{A}(\theta(\vec{i}))$ according to (23). It suffices to show that each read reference $A(f(\vec{i} - \vec{d}))$ is mapped to $\mathcal{A}(\theta(\vec{i}) - \vec{d})$. Three cases are distinguished depending on whether $A(f(\vec{i} - \vec{d}))$ is (a) a local element, (b) a read-only element received from the host, or (c) a nonlocal element received from another processor. In case (a), $A(f(\vec{i} - \vec{d}))$ is mapped to $\mathcal{A}(\theta(\vec{i} - \vec{d}))$ according to (23). By using the assumption that $\forall k : d_k \leq B_k$, we can show by an algebraic manipulation that $\theta(\vec{i} - \vec{d}) = \theta(\vec{i}) - \vec{d}$. In case (b), $A(f(\vec{i} - \vec{d}))$ is read-only, it is mapped $\mathcal{A}(\theta(\vec{i}) - \vec{d})$ directly according to (24). In case (c), due to Lemma 1, the same function η used for mapping the read-only data is used for mapping the nonlocal data. If $A(f(\vec{i} - \vec{d}))$ is a non-local element, it is mapped $\mathcal{A}(\theta(\vec{i}) - \vec{d})$ directly according to (24). ■

7 SPMD Code in Local Address Space

Figure 9 gives the SPMD code that each processor executes in its own local address space.

In the send code, the communication set $\mathcal{C}_{\vec{i}, \vec{\ell}}$ consists of the data elements in the local address space of each sending processor. On arriving at a receiving processor, these data elements are mapped to the local memory of the receiving processor using the function ζ given in (27).

The rewriting of the loop body is straightforward due to Theorem 4.

The tile loops are the same as those in Figure 4.

The element loops are obtained as follows. To avoid expensive conversions from global to local loop indices, an efficient solution is developed to maintain both the global and local loop indices. Note that all n equations in (21) are independent of each other. Therefore, we maintain the original

```

Receive and Unpack the Read-Only Data From the Host ;
for( $t_1 = L_1^t + (p_1 - L_1^t) \bmod P_1$ ;  $t_1 \leq U_1^t$ ;  $t_1 += P_1$ )
...
  for( $t_m = L_m^t + (p_m - L_m^t) \bmod P_m$ ;  $t_m \leq U_m^t$ ;  $t_m += P_m$ )
    for( $t_{m+1} = L_{m+1}^t$ ;  $t_{m+1} \leq U_{m+1}^t$ ;  $t_{m+1} ++$ )
      ...
        for( $t_n = L_n^t$ ;  $t_n \leq U_n^t$ ;  $t_n ++$ )
          /* Receive and Unpack Messages for Tiles from Neighbouring PEs */
          1  for  $\vec{\ell} \in \mathbf{L}$ 
          2    for  $\vec{s} \in \text{pred}(\vec{t}, \vec{\ell})$  in lexicographic order  $\prec$ 
          3      if  $\text{valid}(\vec{s}) = \text{false}$  continue;
          4      if  $\vec{t} = \text{minsucc}(\vec{s}, \vec{\ell})$  then
          5         $\text{rcv}((\vec{p} - \vec{\ell}) \bmod \vec{P}, \text{buf}, \text{sizeof}(\text{buf}))$ ;
          6         $\text{len} := 0$ ;
          7        for  $\vec{i} \in \mathcal{C}_{\vec{s}, \vec{\ell}}$  in lexicographic order  $\prec$ 
          8           $\mathcal{A}(\zeta(\vec{i})) := \text{buf}[\text{len}++]$ ;

           $i_1 = \max(L_1, t_1 B_1 + i_{\min, 1})$ ;
           $u_1 = \min(U_1, (t_1 + 1) B_1 - 1 + i_{\min, 1})$ ;
           $i'_1 = B'_1 \lfloor \frac{t_1}{P'_1} \rfloor + i_1 - t_1 B_1 - i_{\min, 1} + d_{\max, 1}$ ;
          for  $i_1 = (; i_1 \leq u_1; i_1 ++, i'_1 ++)$ 
            ...
             $i_n = \max(L_n, t_n B_n + i_{\min, n})$ ;
             $u_n = \min(U_n, (t_n + 1) B_n - 1 + i_{\min, n})$ ;
             $i'_n = B'_n \lfloor \frac{t_n}{P'_n} \rfloor + i_n - t_n B_n - i_{\min, n} + d_{\max, n}$ ;
            for  $i_n = (; i_n \leq u_n; i_n ++, i'_n ++)$ 
               $\mathcal{A}(i^{\vec{t}}) = F(\mathcal{A}(i^{\vec{t}} - \vec{d}_1), \dots, \mathcal{A}(i^{\vec{t}} - \vec{d}_r))$ ;

          /* Pack and Send Messages for Tiles to Neighbouring PEs */
          1  for  $\vec{\ell} \in \mathbf{L}$ 
          2    if ( $\forall \vec{s} \in \text{succ}(\vec{t}, \vec{\ell}) : \text{valid}(\vec{s}) = \text{false}$ ) continue;
          3     $\text{len} := 0$ ;
          4    for  $\vec{i} \in \mathcal{C}_{\vec{t}, \vec{\ell}}$  in lexicographic order  $\prec$ 
          5       $\text{buf}[\text{len}++] := \mathcal{A}(i^{\vec{t}})$ ;
          6     $\text{send}((\vec{p} + \vec{\ell}) \bmod \vec{P}, \text{buf}, \text{len})$ ;

Pack and Send the Result to the Host ;

```

Figure 9: SPMD code in local address space.

k -th element loop to enumerate the iterations in global indices along that dimension and perform the translation from i_k to i'_k inside the k -th loop. As a first approximation, we get:

$$\begin{aligned}
 &\text{for}(i_k = \max(L_k, t_k B_k + i_{\min, k}); i_k \leq \min(U_k, (t_k + k) B_k - 1 + i_{\min, k}); i_k ++) \\
 &\quad i'_k = B'_k \lfloor \frac{t_k}{P'_k} \rfloor + i_k - t_k B_k - i_{\min, k} + d_{\max, k};
 \end{aligned}$$

where i'_k is a linear function of i_k with coefficient 1. We only need to compute i'_k once at the beginning of the i_k loop and increment i'_k at the end of each iteration. This gives rise to the element loops shown in Figure 9. Note that for the k -th non-distributed dimension, where $B'_k = B_k$ and $P'_k = 1$, we always have ($\forall m < k \leq n : i'_k = i_k - i_{\min, k} + d_{\max, k}$).

```

/*  $\mathcal{A}[0 : 8, 0 : 4]$  declared here */
if  $p_1 = 0$  then offset =  $d_{\max,1} = 1$  else offset =  $-B_1 = -2$  (by (27))
for( $t'_1 = p_1; t'_1 \leq 2; t'_1++$ )
  for( $t'_2 = 0; t'_2 \leq 1; t'_2++$ )
    if  $2t'_1 + p_1 < 1$  continue;
    /* Receive message  $\mathcal{C}_{(2t'_1+p_1-1, t'_2), (1)}$  */
    recv( $(p_1 - 1) \bmod 2$ , buf, sizeof(buf));
    len := 0;
    for( $i' = 4t'_1 + 2p_1; i' \leq 4t'_1 + 2p_1; i'++$ )
      for( $j' = 2t'_2 + 1; j' \leq 2t'_2 + 2; j'++$ )
        buf[len++] =  $\mathcal{A}(i' + \text{offset}, j')$ ;
     $i = \max(1, 4t'_1 + 2p_1 + 1)$ ;
     $u_1 = \min(9, 4t'_1 + 2p_1 + 2)$ ;
     $i' = 3t'_1 + i - 4t'_1 - 2p_1$ ;
    for (;  $i \leq u_1; i++; i'++$ )
       $j = \max(1, 2t'_2 + 1)$ ;
       $u_2 = \min(4, 2t'_2 + 2)$ ;
       $j' = j$ ;
      for (;  $j \leq u_2; j++; j'++$ )
         $\mathcal{A}(i', j') = \mathcal{A}(i', j' - 1) + \mathcal{A}(i' - 1, j' - 1)$ ;
    if  $2t'_1 + p_1 > 3$  continue;
    /* Send message  $\mathcal{C}_{(2t'_1+p_1, t'_2), (1)}$  */
    len := 0;
    for( $i' = 4t'_1 + 2p_1 + 2; i' \leq 4t'_1 + 2p_1 + 2; i'++$ )
      for( $j' = 2t'_2 + 1; j' \leq 2t'_2 + 2; j'++$ )
        buf[len++] =  $\mathcal{A}(i', j')$ ;
    send( $(p_1 + 1) \bmod 2$ , buf, len);

```

Figure 10: The SPMD program for Example 1 in local address space.

The SPMD code can be optimised further by using standard techniques such as constant propagation, common subexpression elimination and loop invariant motion. These optimisations can be performed by a sequential compiler.

By maintaining both global and local loop variables, we have avoided expensive modulo operations that would otherwise occur due to the translation between global and local indices. The modulo operations in the first m tile loops are unavoidable for arbitrary iteration spaces. When the iteration space is rectangular, the modulo operations can be eliminated from the loop bounds and and the floor operations $\lfloor \frac{t_k}{P_k} \rfloor$ can be eliminated by normalising the m outer tile loops to have stride 1 and then performing appropriate variable substitutions.

The program from Figure 7 in local address space is given Figure 10. Note that the tile loops have been normalised. The new tile loop variables are $t'_1 = 2t_1 + p_1$ and $t'_2 = t_2$.

8 Memory Optimisations

If a program has dependence vectors with negative entries, the iteration space must be skewed to make all dependence vectors nonnegative so that a rectangular tiling can be legally applied. In

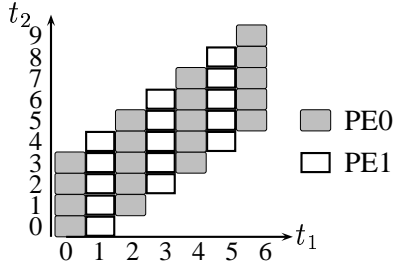


Figure 11: Address folding for a skewed iteration space.

addition, all anti and output dependences must be removed, which may be done by array expansion [14]. Two memory optimisations are developed to reduce the amount of memory usage: *address folding* deals with skewed iteration spaces and *memory recycling* with expanded arrays.

8.1 Address Folding

The basic idea is introduced using an example shown in Figure 11. Suppose A is an array defined in the entire iteration space, where $\mathbf{I} = \{(i, j) \mid 0 \leq i \leq 20 \wedge \max(0, i - 3) \leq j \leq i + 7\}$. This means that $\vec{i}_{\min} = (0, 0)$ and $\vec{i}_{\max} = (20, 27)$. Assume that $\vec{B} = (3, 3)$, $\vec{d}_{\max} = (1, 2)$ and $\vec{P} = (2)$. According to (18), the local array declared will be $\mathcal{A}[0 : 15, 0 : 31]$. Since both the iteration and data spaces are skewed, many elements of \mathcal{A} are not used. The size of the second dimension of this array can be roughly halved if we exploit the fact that any chain of tiles allocated to the same processor consists of at most five tiles. In other words, the number of elements of \mathcal{A} accessed for a fixed value in the first dimension is at most $5 \times B_2 + d_{\max,2} = 5 \times 3 + 2 = 17$. Thus, a smaller local array $\mathcal{A}[0 : 15, 0 : 16]$ can be declared instead. In this two-dimensional case, the address folding can be defined as a mapping from the local indices to *folded local indices*: $(i'_1, i'_2) \mapsto (i'_1, i'_2 \bmod 17)$. The SPMD code can be modified appropriately to reflect this mapping.

In general, let

$$R_k = \max\{|i_k - j_k| + 1 \mid (x, i_k, y), (x, j_k, y) \in \mathbf{I}\} \quad (28)$$

where R_k represents the maximal number of iterations (or *range*) on any line that is within the iteration space and parallel to the k -th elementary vector. Then, the local array declaration $\mathcal{A}_f[0 : u_1^f, \dots, 0 : u_n^f]$ will be declared, where the upper bounds are given by:

$$u_k^f = \begin{cases} \text{if } 1 \leq k \leq m & \rightarrow B'_k \lceil \frac{R_k + B_k}{B_k P_k} \rceil - 1 \\ \square \text{ else} & \rightarrow R_k + d_{\max,k} - 1 \\ \text{fi} \end{cases}$$

where $\lceil \frac{R_k + B_k}{B_k} \rceil$ gives the maximal number of tiles along that dimension in the worst case.

To calculate R_k , it is infeasible if all the points in the iteration space have to be checked. We use the properties of convex polyhedra to reduce the number of points to be checked. Only the vertices

of the iteration space \mathbf{I} need to be checked, as the maximum ranges occur where either or both end points are the vertices, which can be found efficiently for regularly structured computations.

The function that folds the local indices of \mathcal{A} to those of \mathcal{A}_f can be defined as follows:

$$(i'_1, \dots, i'_n) \mapsto (i'_1, \dots, i'_n) \bmod ((u_1^f + 1), \dots, (u_n^f + 1))$$

This mapping is injective by construction.

The modulo operations on i'_k can be coded by exploiting the fact that i'_k increments by 1 at each iteration of the k -th element loop in Figure 9. So $i'_k \bmod (u_k^f + 1)$ is performed only once at the beginning of the loop and set to 0 once it reaches $u_k^f + 1$.

8.2 Memory Recycling

Many applications such as stencil-based computations will require array expansion to be performed on the original programs. Frequently, the outermost dimension of the loop nest would cause a new array dimension, in the case of SOR this would represent time steps. Each hyperplane (constructed from all but the time dimension) would then contain the results at a particular time step. Only the last hyperplane is required for the results. All others are only used to store intermediate results.

Introducing dimensions into the data space increases the amount of memory required. *Memory recycling* is a technique of allocating less memory than required for the complete problem and reusing that memory as the computations proceed. The array needs only to hold enough information to perform its current computations. All previous information is old and can be overwritten.

An important question is: How many hyperplanes do we need to perform the computations if the hyperplanes that were unused are reused for other computations?

Memory recycling consists of defining a recycling function that maps the elements in a local array into an array of a reduced size. It must be designed so that all three types of data elements of an array are handled correctly: (a) the locally computed data, (b) the nonlocally computed data, and (c) the read-only data. The two key observations are as follows:

- All useful data produced in a tile accessed by other processors are sent immediately after the tile is executed (see the send code of Figure 5).
- A message (i.e., a communication set) arriving at a processor will be received and unpacked into the local memory of the processor only just before the first tile that depends on the message is executed (see line 4 of the receive code of Figure 5).

Therefore, if k is an expanded dimension and distributed, we wish to reduce the size of the local array to $B_k + d_{\max,k}$ in that dimension. The local array declared will be $\mathcal{A}_e[0 : u_1^e, \dots, 0 : u_n^e]$, where $u_k^e = B_k + d_{\max,k} - 1$ and the other upper bounds u_i^e , where $i \neq k$, are declared as in (18). This is because the information that is needed for a tile to perform its computations only extends back $d_{\max,k}$ iterations in an expanded dimension. Then, the local indices along those dimensions are simply folded appropriately using the modulo operations:

$$(i'_1, \dots, i'_n) \mapsto (i'_1, \dots, i'_{k-1}, i'_k \bmod (B_k + d_{\max,k}), i'_{k+1}, \dots, i'_n) \quad (29)$$

The locations where the read-only data are stored can also be recycled. Since a processor \vec{p} knows the read-only data it accesses, which are in $\mathbf{O}_{\text{host}, \vec{p}}$ given in (11). Therefore, the SPMD program is modified so that the read-only data are saved in a separate data structure and then used to perform the required initialisations appropriately. This will be explained using the SOR example in Section 9.

The key for memory recycling to work is to ensure that the nonlocal data are accessed correctly.

LEMMA 2 *Let the k -th distributed dimension be recycled. Then $\mathcal{C}_{\vec{s}, \vec{\ell}}$ and $\mathcal{C}_{\vec{s}', \vec{\ell}'}$ are mapped to the same local memory region if and only if $\vec{\ell} = \vec{\ell}'$ and $\forall i \neq k : s_i = s'_i$.*

Proof. Follows from (26) and (29).

THEOREM 5 *Let the expanded dimension k be recycled. Assume that all read-only data are read correctly. The results for an expanded dimension k are computed the same as in the non-recycled case when k is a distributed dimension.*

Proof. It suffices to show that the space for the nonlocal data is not recycled when the nonlocal data will still be accessed. Let tile \vec{t} be the next one to be executed by processor \vec{p} . Let $\vec{s} \in \text{pred}(\vec{t}, \vec{\ell})$ be a valid tile, where $\vec{s} = \vec{t} - \vec{\delta}$ such that $\vec{\delta}[1 : m] = \vec{\ell}$. Note that \vec{s} is executed in the processor $(\vec{p} - \vec{\ell}) \bmod \vec{P}$. We show that the space for the communication set $\mathcal{C}_{\vec{s}, \vec{\ell}}$ will not be recycled before tile \vec{t} is executed. Let \vec{s}' be a tile such that (a) $\vec{s} \prec \vec{s}'$, (b) $\phi(\vec{s}') = (\vec{p} - \vec{\ell}) \bmod \vec{P}$ and (c) $\text{minsucc}(\vec{s}', \vec{\ell}) \prec \vec{t}$. By Lemma 2, it suffices to show that the communication set $\mathcal{C}_{\vec{s}', \vec{\delta}}$ will not be stored where the communication set $\mathcal{C}_{\vec{s}, \vec{\delta}}$ is. When (a) – (c) are true, the first m entries of \vec{s}' must be $(s'_1, \dots, s'_m) = (t_1 - \ell_1, \dots, t_m - \ell_m)$. Since the k -th dimension is distributed, where $1 \leq k \leq m$, the spaces for storing both $\mathcal{C}_{\vec{s}', \vec{\ell}}$ and $\mathcal{C}_{\vec{s}, \vec{\ell}}$ cannot overlap according to Lemma 2. ■

A non-distributed expanded dimension can also be recycled if it is the only non-distributed dimension or if the data dependences in the program exhibit some desired patterns. In this case, the local indices along the recycled non-distributed dimension k can be folded as follows

$$i'_k \mapsto \begin{cases} \text{if } i'_k < d_{\max, k} & \rightarrow i'_k \\ \text{fi} & \rightarrow (i'_k - d_{\max, k}) \bmod B_k + d_{\max, k} \end{cases}$$

In addition, the data produced in a tile along the k -th dimension must be copied to where the read-only data are stored in the k -th dimension.

Memory recycling is significant because often the expanded dimension represents time steps. By applying memory recycling, the size of the local array along the time dimension depends on the size of tiles in that dimension not the number of time steps the program is executed.

The performance results of the two memory optimisations are presented in Figure 13(b).

9 Experimental Results

The objectives of our experiments are twofold. One is to evaluate all proposed compiler techniques and the other is to identify some performance-critical factors for running tiled code on distributed

memory machines. Therefore, SOR serves as a good example to achieve the two objectives. All experiments were carried out on a Fujitsu AP1000 consisting of 128 SPARC cells each consisting of a SPARC processor running at 25MHz with 16MB RAM. All processors are connected by a 2D torus network called the *T-net*. The T-net provides the interprocessor communication using the wormhole routing with a 25MB/sec of bandwidth. All I/O is performed on the host.

We start with the following five-point SOR program:

$$\begin{aligned}
& \text{for}(t = 0; t \leq M - 1; t++) \\
& \quad \text{for}(i = 1; i \leq N; i++) \\
& \quad \quad \text{for}(j = 1; j \leq N; j++) \\
& \quad \quad \quad a(i, j) = \frac{\omega}{4}(a(i-1, j) + a(i, j-1) + a(i+1, j) + a(i, j+1)) + (1-\omega)a(i, j)
\end{aligned} \tag{30}$$

where $a[0 : N + 1][0 : N + 1]$ is a 2D array. The dependences for the program are: $\mathbf{D}_a = \{(1, 0, 0), (0, 1, 0), (1, -1, 0), (1, 0, -1), (0, 0, 1)\}$, where $(1, 0, 0)$ is caused due to a self output dependence on $a(i, j)$. Next, we eliminate this output dependence by expanding a into a 3D array $A[0 : M, 0 : N + 1 : 0 : N + 1]$ as follows:

$$\begin{aligned}
& \text{for}(t = 1; t \leq M; t++) \\
& \quad \text{for}(i = 1; i \leq N; i++) \\
& \quad \quad \text{for}(j = 1; j \leq N; j++) \\
& \quad \quad \quad A(t, i, j) = \frac{\omega}{4}(A(t, i-1, j) + A(t, i, j-1) + A(t-1, i+1, j) + A(t-1, i, j+1)) + (1-\omega)A(t-1, i, j)
\end{aligned}$$

The read-only data of A are initialised as follows:

$$A[t][i][j] = \begin{cases} \text{if } t = 0 & \rightarrow a(i, j) \\ \quad \square \quad i = 0 & \rightarrow a(0, j) \\ \quad \square \quad i = N + 1 & \rightarrow a(N + 1, j) \\ \quad \square \quad j = 0 & \rightarrow a(i, 0) \\ \quad \square \quad j = N + 1 & \rightarrow a(i, N + 1) \\ \text{fi} \end{cases} \tag{31}$$

The results of the computations can be found in $A[M][i][j]$.

We apply $\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}$ to skew the expanded program to make all dependences nonnegative:

$$\begin{aligned}
& \text{do } t' = 1, M \\
& \quad \text{do } i' = t' + 1, t' + N \\
& \quad \quad \text{do } j' = 2t' + 1, 2t' + N \\
& \quad \quad \quad t = t', i = -t' + i', j = -2t' + j'; \\
& \quad \quad \quad A(t, i, j) = \frac{\omega}{4}(A(t, i-1, j) + A(t, i, j-1) + A(t-1, i+1, j) + A(t-1, i, j+1)) + (1-\omega)A(t-1, i, j)
\end{aligned}$$

The dependence set becomes: $\mathbf{D}_A = \{(1, 1, 2), (0, 1, 0), (1, 0, 2), (1, 1, 1), (0, 0, 1)\}$. To make all dependences nonnegative, it is sufficient to replace the 2 in the skewing matrix is replaced with 1. However, skewing the inner two loops with different skewing factors creates more irregular iteration spaces so that the performance of our compiler techniques can be tested more thoroughly.

At this point, we calculate that $\vec{i}_{\min} = (1, 2, 3)$, $\vec{i}_{\max} = (N, M + N, 2M + N)$ and $\vec{d}_{\max} = (1, 1, 2)$. The skewed iteration space is tiled with rectangles of size $\vec{B} = (B_1, B_2, B_3)$ and the tiled code is obtained per Section 2.3. The first two dimensions of the three tile loops are distributed cyclically to a 2D processor mesh. The SPMD code is then generated following Figure 9. Since the skewed iteration space is no longer rectangular — one of the main reasons why SOR is used in our

	RECYCLING	NO RECYCLING
dim 1 (u_1)	B_1+1	$(B_1+1)\lceil\frac{M+1}{B_1P_1}\rceil-1$

	FOLDING	NO FOLDING
dim 2 (u_2)	$(B_2+1)\lceil\frac{N+B_2}{B_2P_2}\rceil-1$	$(B_2+1)\lceil\frac{M+N-1}{B_2P_2}\rceil-1$
dim 3 (u_3)	$B_3\lceil\frac{N+B_3}{B_3}\rceil+1$	$B_3\lceil\frac{2M+N-2}{B_3}\rceil+1$

Figure 12: Declaration of local array $\mathcal{A}[0 : u_1, 0 : u_2, 0 : u_3]$.

experiments, the general message-passing code for arbitrary iteration spaces in Figure 5 is adopted. The message-passing code for I/O is generated as discussed at the beginning of Section 5.

The speedup of an SPMD program is measured as:

$$S_p = \frac{T_s}{T_p}$$

where T_s is the execution time of the (untiled) sequential program given in (30) on a single AP1000 processor and T_p is the execution time of the SPMD program on p AP1000 processors.

Because we are dealing with DOACROSS loop nests with dependences spanning all dimensions of the iteration space, 100% speedup is not attainable due to unavoidable communication cost. To appreciate this, we give below a conservative speedup upper bound for the SOR's SPMD code in the special case when all processors are assigned exactly the same number of iterations:

$$S_p \leq \frac{MN^2t_c}{\frac{MN^2}{P_1P_2}t_c + (P_1 - 1)(\alpha + 4B_2B_3\beta) + (P_2 - 1)(\alpha + 4B_1B_3\beta)} \quad (32)$$

where t_c is the computation time for one iteration, α is the communication startup cost and β is the time for transferring one byte between two processors. On AP1000, we have measured previously that $\alpha = 0.00037$ secs and $\beta = 0.00003$ secs [9]. In the case of the five-point SOR, $t_c = 0.0000028$ secs.

Therefore, in (32), the nominator represents $T_s = MN^2t_c$. The denominator is derived as follows. Due to the tile dependences $\Delta = \{(0, 1), (1, 0), (1, 1)\}$, the processor $(P_1 - 1, P_2 - 1)$ is always the last finishing processor. The first tiles executed in the following processors form a chain of dependent tiles: $(0, 0) \rightarrow \dots \rightarrow (0, P_2 - 1) \rightarrow (1, P_2 - 1) \rightarrow \dots \rightarrow (P_1 - 1, P_2 - 1)$. As a result, a processor cannot start unless its predecessor in the chain has executed its first tile. Each of the first P_2 (last P_1 , resp.) processors sends B_1B_3 (B_2B_3 , resp.) elements (4 bytes per element) produced in the first tile it computes to the next processor in the chain. So the total communication time elapsed along the chain is the sum of the last two terms in the denominator of (32). Assume conservatively that once the last processor $(P_1 - 1, P_2 - 1)$ starts, computation and communication will be completely overlapped. But still this processor takes $\frac{MN^2}{P_1P_2}t_c$ (the first term in the denominator) time to execute sequentially the $\frac{MN^2}{P_1P_2}$ allocated iterations. Hence, the best speedup possible in this special case is given by (32).

Our experimental results are given in Figure 13. We describe the rationale behind each experiment and draw conclusions where appropriate.

Tiling Cost If a loop nest is skewed before being tiled, more complex loop bounds in the skewed code introduce into the tiled code even more complex loop bounds involving min and max. To measure its effect on the execution time, we have designed a test case, presented in Figure 13(a), that achieves the perfect balance such that each processor gets exactly $\frac{M \times N \times N}{P_1 P_2} = 320,000$ iterations. To reduce the impact of boundary tiles and due to the memory limit, we have used only $P_1 \times P_2 = 5 \times 5 = 25$ processors. The two memory optimisations are not used so that their interference is avoided. In this test case, the sequential execution time is $T_s = 22.37$ secs. To measure the time spent only on the computations, we comment out all send and receive calls in the SPMD program, assuming ideally 0 as the communication cost. The parallel execution time on 25 processors is measured to be $T_p = 1.05$ secs. This imposes an upper limit $\frac{22.37}{1.05} \approx 21.30$ on the best speedup possible. (Recall that T_s in the speedup formula (32) is the sequential execution time of the untiled version)

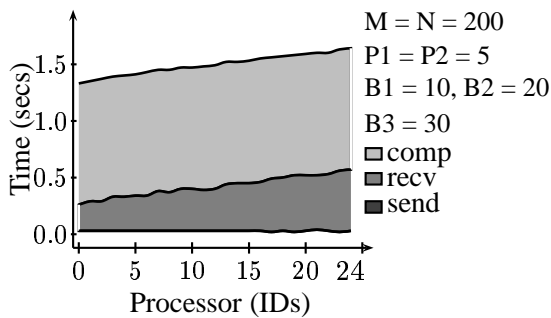
While the tiled code can be optimised by such optimisations as common subexpression elimination and loop invariant motion, it may be worthwhile considering special-purpose techniques for optimising the tiled code.

Communication Cost Our communication scheme is efficient since it combines small messages into large messages, which encompasses the three classic optimisations: message vectorisation, coalescing and aggregation [19]. Consider again the test case shown Figure 13(a) we analysed above, where all processors are given exactly 320,000 iterations and $T_s = 22.37$ secs. The parallel execution time is 1.65 secs. This translates to a speedup of $\frac{22.37}{1.65} = 13.56$ on 25 processors. According to (32), the best speedup possible is only 16.44.

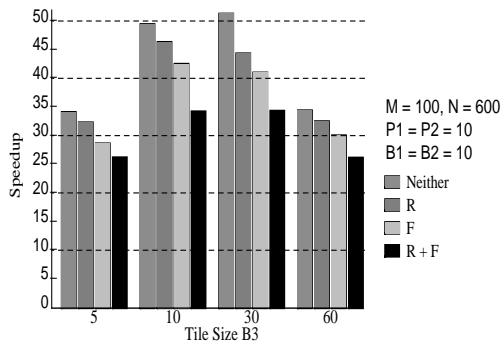
Validity Test Due to the cyclic computation and data distribution, a processor is required to test at run time whether some tiles are valid or not (line 2 of the send code and lines 2 – 4 of the receive code in Figure 5). By applying the optimisations discussed in Section 5.2, the time elapsed on this part of the program is within 3% of the total execution time among all experiments we have conducted and is thus negligible.

Memory Optimisations The time dimension is distributed and can be recycled according to Theorem 5. To fold the other two dimensions, note that the number of iterations along each of the two dimensions is $R_2 = R_3 = N$ according to (28). This leads to Figure 12. Address folding is very useful for skewed iteration space. In the case when $M = N$, for example, the local memory allocated is reduced to one sixth. Memory recycling is crucial for expanded arrays. By recycling the time dimension, the size of the local array in that dimension is $B_1 + d_{\max,1} = B_1 + 1$, which is independent of M , the number of time steps used. However, future research is still needed to reduce the memory cost even further.

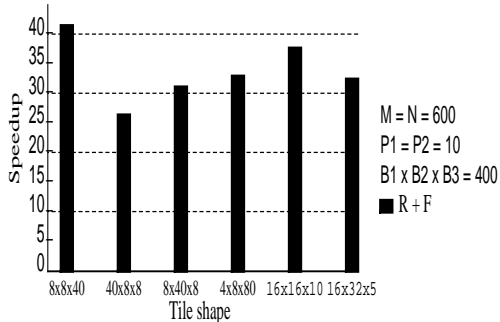
In our SPMD mode of execution, all read-only data are initially received before the tiles are executed. When a dimension is recycled, the locations where the read-only data are supposed to be stored permanently can be recycled, too. Therefore, these read-only data are saved in a separate array and then used to provide the required initialisation. In the SOR example, the read-only data are specified in (31).



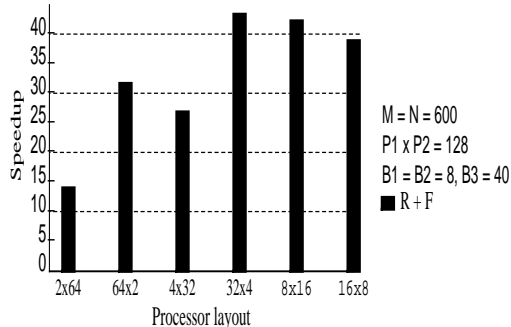
(a) Impact of tiling and comm cost



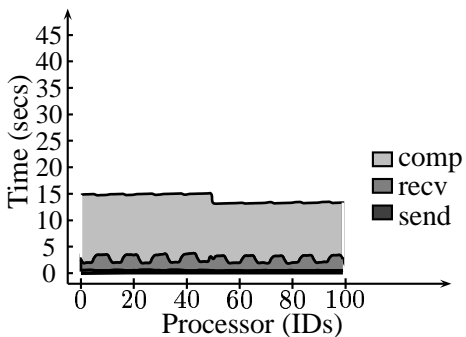
(b) Impact of memory optimisations



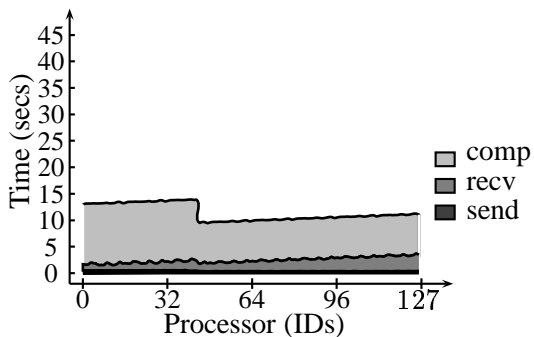
(c) Impact of tile shape on performance



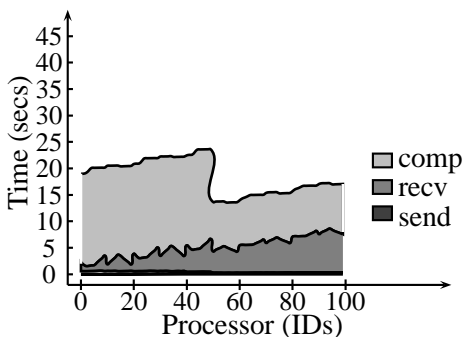
(d) Impact of processor layout on performance



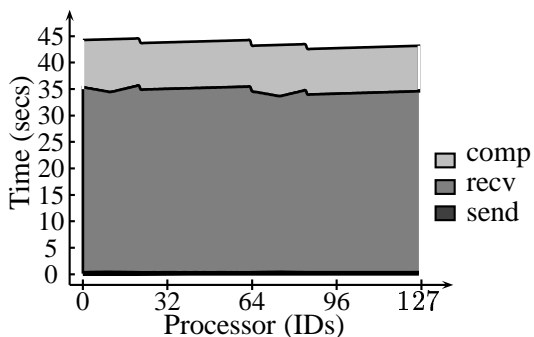
(e) Case $8 \times 8 \times 40$ in (c)



(f) Case 32×4 in (d)



(g) Case $40 \times 8 \times 8$ in (c)



(h) Case 2×64 in (d)

Figure 13: Experimental results of SOR on AP1000.

As shown in Figure 13(b), the memory space saved by the two optimisations is achieved at the expense of CPU cycles. While further optimisations may be sought for division and modulo operations, the overhead is expected to be small for large computation tasks.

Granularity (i.e., Computation and Communication Overlap) Figure 13(b) also serves to illustrate the importance of reducing communication overhead by overlapping computation and communication. In all four test cases, the tile sizes B_1 and B_2 in the two distributed dimensions are fixed, implying that each processor has the same workload. By adjusting the tile size B_3 in the non-distributed dimension, we change the granularity of tiles. The performance varies mainly due to the degree at which computation and communication is overlapped. The smaller the B_3 is, the larger the parallelism is but the larger the communication cost will be. The converse is true as B_3 is gradually increased. Previous work on determining tile sizes [26, 29, 30, 38] usually assumes rectangular iteration spaces with nonnegative dependence vectors and does not consider a concrete computation distribution on a specific machine.

Load Balance Poor performance usually results if the processors do not have balanced workload. We ran the program by fixing the granularity of a tile at $B_1 \times B_2 \times B_3 = 400$ and the processor layout at $P_1 \times P_2 = 10 \times 10$. We change the workload of a processor by adjusting the aspect ratio of a tile, which affects the performance as shown in Figure 13(c). The best speedup among the six test cases is obtained when all processors compute roughly the same number of iterations as shown in Figure 13(e). In the worst case, as displayed in Figure 13(g), the first 50 processors are assigned about twice as many iterations as the other 50 processors due to the tile size $B_3 = 40$ used.

Processor Assignment and Parallelism On the other hand, we are not guaranteed to achieve good performance even if all processors are assigned the same workload. Figure 13(d) indicates the sensitivity of the performance to the processor layout given a fixed number of processors. In both the best and worst of the six test cases displayed in Figures 13(f) and (h), respectively, all processors are assigned roughly the same number of iterations. However, in the worst case shown in Figure 13(h), all processors spend about 80% of the total run time waiting at the receiving statement. But the speedup more than doubles when the two processor dimensions are swapped (see the second bar in Figure 13(d)). This is because the second distributed dimension is swept in the second tile loop in each processor. When this dimension is distributed to 64 processors, the first few processors are given $\lceil \frac{N+B_2}{B_2 \times P_2} \rceil = \lceil \frac{600+8}{8 \times 64} \rceil = 2$ tiles along that dimension. These few processors cannot execute any tiles along the first dimension unless they have executed these two tiles. This has almost serialised the execution of the entire program.

10 Related Work

Given a tiling transformation, this paper presents compiler techniques for generating a SPMD program to execute a rectangularly tiled iteration space. Some closely related work is reviewed below.

Previous work on tiling includes: improving the performance of a memory hierarchy [6, 7, 23, 25, 35, 40], determining the sizes and shapes of tile to minimise communication overhead on distributed memory machines [10, 29, 30, 33, 38] and determining the tile size to minimise execution

time on distributed memory machines [1, 5, 26]. To integrate our techniques with a data-parallel compiler, compiler techniques for selecting a tiling transformation for commonly occurring iteration space shapes and for selecting a processor layout remain to be developed. We are not aware of any work studying the impact of processor layout on the performance of tiled code. Almost all previous work on time-minimal tilings focuses on finding good rectangular tiles for rectangular iteration spaces without taking cache optimisation on a single processor into account [1, 5, 26]. The problem of selecting a tiling that minimises the total execution time by considering many factors simultaneously such as communication overhead and cache performance is difficult. Some initial attempt can be found in [25].

Methods for removing anti and output dependences and for transforming programs into single assignment form are many: array expansion [14], node splitting [27], array privatisation [17] and others [4]. Recently, a partial array expansion technique is proposed to reduce the amount of memory usage [24]. Removing anti and output dependences for SOR-like programs with an outmost time loop is simple.

General techniques for compiling data-parallel languages can be found in [2, 19] and the survey paper [11]. In compiling HPF-like data parallel languages, the data distribution is usually specified by the programmer. Essentially, the data space of an array is first tiled by rectangular tiles and the tiles are then distributed cyclically to the processors for some dimensions of the data space and collapsed to the same processor in the other dimensions. After the data distribution is performed, the computation distribution is inferred by the compiler. In compiling tiled code, these two phases are reversed in this paper. When a tile is treated as an atomic unit of computation, the computation distribution is partially specified. Therefore, the approach discussed in this paper first determines the computation distribution and then infers the data distribution. When a rectangular tiling is applied to a skewed iteration space, the data space of an array can be viewed as being tiled by parallelepipeds which are not necessarily rectangles. Since the program has single-assignment semantics after array expansion, the owner-computes rule is still enforced.

After both data distribution and computation distribution are performed, message-passing code must be generated. The problem of deriving communication sets is addressed in [8, 18, 21, 34]. In [11], three approaches for enumerating regular communication sets are identified: array sections, finite state machines, and polyhedra and lattices. This paper manipulates polyhedra represented by affine constraints and generates a set of loops to enumerate a message approximated by an array section. Our communication scheme naturally encompasses such optimisation techniques as message vectorisation, coalescing and aggregation for optimising communication as discussed in [19]. Several storage schemes for managing nonlocal data, such as buffers, hash tables, software paging and overlap areas are discussed in [11, 15]. This paper uses an extended concept of overlap areas to store the nonlocal data.

Our earlier paper [33] addresses the communication code generation for tiled code in the special case when the tiled iteration spaces are rectangular. Our technical report [41] contains some preliminary ideas of address folding and memory recycling. Our previous results on measuring the performance of AP1000 can be found in [9, 33].

The Omega Calculator [28] has been particularly useful for generating packing and unpacking code given a set of inequalities described using Presburger formulas.

11 Conclusions

In this paper, we have presented compiler techniques for generating an SPMD program for efficient execution on a distributed memory machine given a tiled iteration space. We use a cyclic computation distribution to allocate tiles to processors to reduce load imbalance and overlap computation and communication. We use the computer-owns rule to derive the data distribution from the computation distribution. We provide techniques for generating all required communication code. In particular, we have presented efficient message-passing code for both arbitrary and rectangular iteration spaces. We introduce a storage scheme for managing both the local and nonlocal data. By extending the concept of overlap areas slightly, our SPMD code exhibits high locality of references despite of any sparse array references used in the program. We have developed two memory optimizations to reduce the amount of memory usage for skewed iteration spaces and expanded arrays. We provide all the require translation functions between global and local indices. We present a complete SPMD code that a processor executes in its own local address space. By maintaining both global and local loop variables, we have avoided expensive runtime translation between and global local indices.

We have evaluated and validated all compiler techniques in a Fujitsu AP1000. Our performance results indicate that tiling is a useful performance-improving optimisation for distributed memory machines and our compiler techniques can generate efficient SPMD programs for tiled iteration spaces. We have identified several research problems that need to be further investigated. These include the problem of generating tiled code with efficient loop structure and loop bounds and the problem of selecting tile sizes and shapes to overlap computation and communication and to minimise load imbalance.

While presented in the context of compiling for one single loop nest, our compiler techniques are general enough to be extended for multiple loop nests. To integrate these techniques with a data-parallel compiler, it is desirable to choose a tiling transformation in such a way that the implied data distribution corresponds to the user-specified data distribution. This may be achieved in combination with an appropriate choice of the tile allocation function discussed in Section 3, as one of the referees suggests. This topic will be investigated in future work.

12 Acknowledgements

We are grateful to Australian National University for providing us the access to their Fujitsu AP1000. We also wish to thanks the referees for their comments and suggestions. The first author acknowledges gracefully the support of an Australian Research Council Grant A49232251. The second author would like to thank Professor David Padua of University of Illinois at Urbana-Champaign for discussions during his sabbatical at Urbana-Champaign. The second author also gracefully acknowledges the support of an Australian Research Council Grant A49600987.

References

- [1] R. Andonov and S. Rajopadhye. Optimal tiling of two-dimensional uniform recurrences. Technical Report 97-01, LIMAV, Université de Valenciennes, Jan. 1997.

- [2] S. Benkner, B. M. Chapman, and H. P. Zima. Vienna fortran 90. In *Scalable High Performance Computing Conference*, pages 51–59, Apr. 1992.
- [3] V. Bouchitté, P. Boulet, A. Darte, and Y. Robert. Evaluating array expressions on massively parallel machines with commuication/computation overlap. *Int. Journal of Supercomputer Applications and High Performance Computing*, 9(3):205–219, 1995.
- [4] P. Y. Calland, A. Darte, Y. Robert, and F. Vivien. On the removal of anti and output dependences. Technical Report 96–04, Ecole Normale Supérieure de Lyon, Feb. 1996.
- [5] P. Y. Calland, J. Dongarra, and Y. Robert. Tiling with limited resources. Technical Report 97–??, Ecole Normale Supérieure de Lyon, Feb. 1997.
- [6] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing '92*, pages 114–124, Minneapolis, Minn., Nov. 1992.
- [7] L. Carter, J. Ferrante, and S. Flynn Hummel. Efficient parallelism via hierarchical tiling. In *SIAM Conf. on Parallel Processing for Scientific Programming*, New York, 1995.
- [8] Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shung-Hua Teng. Generating local addresses and communication sets for data-parallel programs. In *4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 149–158, San Diego, Calif., May 1993.
- [9] S. Chen and J. Xue. Issues of tiling double loops on distributed memory machines. In *Proc. 5th Australian Parallel and Real-Time Systems*, pages 377–388, 1998.
- [10] Y.-S. Chen, S.-D. Wang, and C.-M. Wang. Tiling nested loops into maximal rectangular blocks. *J. of Parallel and Distributed Computing*, 35(2):108–120, 1996.
- [11] F. Coelho, C. Germain, and J. L. Pazat. State of the art in compiling hpf. In G. R. Perrin and A. Darte, editors, *Data Parallel Programing Model: Foundations, HPF Realization and Scientific Applications*, Lecture Notes in Computer Science 1132, pages 659–664. Elsevier (North-Holland), 1996.
- [12] J.-F. Collard, T. Risset, and P. Feautrier. Construction of DO loops from systems of affine constraints. Technical Report 93-15, Ecole Normale Supérieure de Lyon, May. 1993.
- [13] A. Darte and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. Technical Report 96–34, Ecole Normale Supérieure de Lyon, November 1996.
- [14] P. Feautrier. Dataflow analysis for array and scalar references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.
- [15] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, 1990.
- [16] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 15–29, Toronto., Jun. 1991.
- [17] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing '95*. ACM Press, 1995.
- [18] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, Chua-Huang Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *1993 International Conference on Parallel Processing*, volume II, pages 301–305, St. Charles, Ill., August 1993.

- [19] S. Hiranandani, K. Kennedy, and C. W.-Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.
- [20] F. Irigoin and R. Triolet. Supernode partitioning. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California., Jan. 1988.
- [21] C. Koelbel. Compile-time generation of regular communication patterns. In *Supercomputing '91*, pages 101–110. ACM Press, 1991.
- [22] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, 1994.
- [23] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, Apr. 1991.
- [24] V. —Lefebvre and P. Feautrier. Optimizing storage size for static control prgrams in automatic parallelizers. In *1997 European Parallel Processing Conference*, 1997.
- [25] N. Mitchell, L. Carter, J. Ferrante, and K. Hogstedt. Quantifying the multi-level nature of tiling interactions. *Int. Journal of Parallel Programming*, 1998. Earlier version at 10th Workshop on Languages and Compilers for Parallel Computing.
- [26] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjustment in compiling for general DOACROSS loop nests. In *1995 ACM International Conference on Supercomputing*, pages 270–279. ACM Press, 1995.
- [27] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. ACM*, 29(12):1184–1201, Dec. 1986.
- [28] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, 35(8):102–114, Aug. 1992.
- [29] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *J. of Parallel and Distributed Computing*, 16(2):108–230, Oct. 1992.
- [30] R. Schreiber and J. J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, RIACS, May 1990.
- [31] A. Schrijver. *Theory of Linear and Integer Programming*. Series in Discrete Mathematics. John Wiley & Sons, 1986.
- [32] W. Shang and Jose A. B. Fortes. Independent partitioning of algorithms with uniform dependencies. *IEEE Trans. on Computers*, 41(2):190–206, Feb. 1992.
- [33] P. Tang and J. N. Zigman. Reducing data communication overhead for DOACROSS loop nests. In *1994 ACM International Conference on Supercomputing*, pages 44–53. ACM Press, 1994.
- [34] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs using integer lattices. In *9th Workshop on Languages and Compilers for Parallel Computing*, pages 291–301, Aug 1996.
- [35] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*. ACM, Jun. 1991.

- [36] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.
- [37] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
- [38] J. Xue. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.
- [39] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.
- [40] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. *Int. Journal of Parallel Programming*, 26(6):671–696, 1998.
- [41] J. N. Zigman and P. Tang. Implementing global address space in distributed memory machines. Technical Report TR-CS-94-10, Department of Computer Science, The Australian National University, Oct. 1994.