Reducing Data Communication Overhead for DOACROSS Loop Nests

Peiyi Tang and John N. Zigman

Technical Report
ANU-TR-CS-93-16
December 23, 1993

Reducing Data Communication Overhead for DOACROSS Loop Nests

Peiyi Tang and *John N. Zigman*

Department of Computer Science
The Australian National University
Canberra ACT 2601 Australia

December 23, 1993

Abstract

If the loop iterations of a loop nest cannot be partitioned into independent sets, the data communication for data dependences are inevitable in order to execute them on parallel machines. This kind of loop nests are referred to as DOACROSS loop nests.

This paper is concerned with compiler algorithms for parallelizing DOACROSS loop nests for distributed-memory multicomputers. We present a method that combines loop tiling, chain-based scheduling and indirect message passing to generate efficient message-passing parallel codes. We present our experiment results on Fujitsu AP1000 which show that low communication overhead and high speedup for DOACROSS loop nests on multicomputers can be achieved by tuning these techniques.

Keywords: DOACROSS loop nests, chain-based scheduling, loop tiling, indirect message passing, distributed-memory multicomputers

[*]This work was supported in part by **the Australian Research Council** under Grant No. A49232251 and **Fujitsu Laboratories, Ltd.**, Japan

1 Introduction

If the loop iterations of a loop nest cannot be partitioned into independent sets, the data communication for data dependences are inevitable in order to execute the loop nest on parallel processors. This kind of loop nests are known as DOACROSS loop nests.

An example of DOACROSS loop nest is shown in Figure 1. Although the four loops in the loop nest can be interchanged, it can never be transformed to a loop nest with outermost DOALL loops.[1] Many loop nests for solving differential equations using the finite difference method are DOACROSS loop nests.

This paper is concerned with compiler algorithms to generate efficient parallel codes for DOACROSS loop nests on multicomputers. Distributed-memory multicomputer systems such as Fujitsu AP1000 and TMC CM-5 rely on message passing (point-to-point and aggregate) for interprocessor data communication and synchronization. The overhead of message passing in multicomputers is large.

To generate efficient parallel codes for DOACROSS loop nests on multicomputers, the time delay caused by the large overhead of message passing must be reduced. This paper presents a method that combines chain-based scheduling [1], loop tiling [2,3] and indirect message passing to reduce the data communication overhead.

```
for $i_1 = 0$ to 127
    for $i_2 = 0$ to 127
        for $i_3 = 0$ to 127
            for $i_4 = 0$ to 127
                a($i_1+1$,$i_2+1$,$i_3+1$,$i_4+1$) =
                  0.25*(a($i_1+1$,$i_2+1$,$i_3$,$i_4+1$) + a($i_1+1$,$i_2$,$i_3+1$,$i_4+1$) +
                        a($i_1+1$,$i_2+1$,$i_3$,$i_4$) + a($i_1+1$,$i_2$,$i_3$,$i_4$) +
                        a($i_1$,$i_2+1$,$i_3$,$i_4$) + a($i_1+1$,$i_2$,$i_3$,$i_4$) +
                        a($i_1$,$i_2+1$,$i_3+1$,$i_4+1$) + a($i_1+1$,$i_2+1$,$i_3+1$,$i_4+1$) );
```

Figure 1. An Example of DOACROSS Loop Nest

The wave-front method was proposed to parallelize DOACROSS loop nests for vector machines [4,5]. More recently, Wolf and Lam [3] showed that any DOACROSS loop nest can be transformed to a loop nest with DOALL loops enclosed in the outermost serial DO loop. The code of the transformed loop nest generated for multicomputers is of the following form:

[1]A DO loop is a DOALL loop if there are no data dependences between its iterations and, therefore, they can be executed independently in parallel.

```
do loop
    doall loops
        ⋮
        loop body computation;
        ⋮
    enddoall;
    barrier;
    aggregate data communication;
enddo
```

Obviously, this wave-front method on multicomputers does not provide the overlap between computation and data communication. It also requires barrier synchronization to enforce DOALL loops enclosed in the outermost serial loop. The barrier synchronization could be very expensive in large systems. The chain-based scheduling of DOACROSS loop nests [1] allows the data communication to be overlapped with the computation and eliminates the barrier synchronization.

To amortize the large startup overhead of message passing, we also incorporated loop tiling [2,3] into the chain-based scheduling. By tuning the size of the tiles, we can control the granularity of the parallel tasks to reduce the impact of message-passing overhead.

To further reduce the data communication overhead, we invent a novel message packing scheme called *indirect message passing*. The indirect message passing reduces the number of messages without loss of parallelism of the chain-based parallel codes.

To show the effectiveness of these techniques we parallelized and coded the DOACROSS loop nest in Figure 1 and run it on a Fujitsu AP1000 with different configurations of processor array and loop tiling. On the 128-processor AP1000, we can get the speedup as high as 121.9 by tuning the tile size and the shape of processor array.

In Section 2, we describe the loop transformation for loop tiling and chain-based scheduling. In Section 3, we describe the algorithms to generate messages for data communication. In Section 4, we present the method of indirect message passing. The experiment results on AP1000 are presented in Section 5. Section 6 completes the paper with conclusions.

2 Loop Tiling and Chain-Based Scheduling

In this section, we first give the definition of DOACROSS loop nests. Then we present the algorithms for loop tiling and chain-based scheduling.

2.1 Tiling Iteration Space of DOACROSS Loop Nests

The index set of an n-dimensional loop nest with loop steps 1 can be modeled by integer grids within a convex polyhedron determined by the loop bounds. For example, the index set of the loop nest in Figure 1 can be represented by

$$I^4 = \{\vec{i} \in Z^4 | C\vec{i} \leq \vec{c}\}$$

where

$$\mathrm{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \quad and\ \vec{c} = \begin{bmatrix} 127 \\ 127 \\ 127 \\ 127 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

In general, the loop bounds can be affine functions of index variables of the enclosing loops and each loop except the outermost one can have more than one upper or lower bounds. The matrix C has the row-echelon shape if the rows for the same loop are grouped together.

There is a *data dependence* from index $\vec{i_1} \in I^n$ to $\vec{i_2} \in I^n$ if (1) $\vec{i_1}$ is lexicographically less than $\vec{i_2}$, (2) a data element is accessed by both iterations, $\vec{i_1}$ and $\vec{i_2}$ with at least one access is write operation and (3) there is no index $\vec{i_3}$ between $\vec{i_1}$ and $\vec{i_2}$ that writes to the same data element. The vector $\vec{d} = \vec{i_2} - \vec{i_1}$ is called *distance vector*. The dependence distance vector $\vec{d}$ is *constant* if the dependence exits for every pair of $\vec{i_1}, \vec{i_2} \in I^n$ such that $\vec{d} = \vec{i_2} - \vec{i_1}$. In this paper, we only consider the loop nests with constant dependence distance vectors. The distance vectors form a distance matrix:

$$D = \left[\vec{d_1}, \cdots, \vec{d_m}\right]$$

where each $\vec{d_j}$ is a distance vector. The loop nest in Figure 1 has 7 distance vectors and it's distance matrix is

$$\mathrm{D} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

We use the rank of distance matrix D to distinguish DOALL and DOACROSS loop nests.

Definition 1 (DOACROSS **Loop Nest**) *A loop nest with n loops is a* DOACROSS *loop nest if the rank of its dependence distance matrix is n; otherwise, the loop nest is a* DOALL *loop nest.*

The rationale behind Definition 1 is that if the rank of D is less than n, the loop nest can be transformed to have at least one outermost DOALL loops. If the rank of the matrix of distance vectors is n, it is still possible to convert it to a loop nest with outermost DOALL loops. But, the loop range of these DOALL loops in this case are limited by the distance vectors which are very small in real programs. This limited DOALL parallelism may not be sufficient for large parallel systems. In this paper, we simply put it in the category of DOACROSS loop nests.

The number of data elements generated in one iteration to be used in another iteration of a DOACROSS loop nest is usually very small. Therefore, the first step to reduce the run-time data communication overhead is to group a number of iterations to form a larger task and generate larger messages at the end of each task. The technique to do that is called *loop tiling*, an extension of strip-mining transformation in vectorizing compilers. The general theory for loop tiling can be found in [2]. For the purpose of this paper, the simple rectangular tiling is sufficient.

Let us use

$$DO(i_1, \cdots, i_n)((L_1, U_1), \cdots, (L_n, U_n))\{BODY\}$$

to denote a n-dimensional loop nest with L_j and U_j being the lower and upper bounds of the j-th loop of index i_j, $1 \le j \le n$. The index set of the loop nest is

$$I^n = \{\vec{i} \in Z^n | C\vec{i} \le \vec{c}\}$$

where $\vec{i} = (i_1, \cdots, i_n)$[2] and matrix C and vector $\vec{c}$ which define the convex polyhedron are derived from the loop bounds $((L_1, U_1), \cdots, (L_n, U_n))$. A *tile* with *origin* $\vec{i_0}$ is the subset of the index set

$$T(\vec{i_0}) = \{\vec{i} \in I^n | \vec{0} \le \vec{i} - \vec{i_0} \le \vec{k} - \vec{1}\}$$

where $\vec{k} = (k_1, \cdots, k_n)$ is the *size vector* of the tiling and each k_j $(1 \le j \le n)$ is a positive integer. The origins $\vec{i_0}$ form a lattice in the index set and can be generated by an integer matrix L:

$$\vec{i_0} = L\vec{t}$$

In the simple rectangular tiling, it is

$$L = \begin{bmatrix} k_1 & 0 & \cdots & 0 \\ 0 & k_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & k_n \end{bmatrix}$$

and $\vec{t} = (t_1, \cdots, t_n)$ is called *tile index* of the corresponding tile.

Since $L\vec{t} = (k_1 t_1, \cdots, k_n t_n)$, another form for the origin $\vec{i_0}$ is $\vec{i_0} = \vec{k} \cdot \vec{t}$. Therefore the tile with tile index $\vec{t}$, denoted by $T_{\vec{t}}$, is

$$T_{\vec{t}} = T(\vec{k} \cdot \vec{t}) = \{\vec{i} \in I^n | \vec{k} \cdot \vec{t} \le \vec{i} \le \vec{k} \cdot \vec{t} + \vec{k} - \vec{1}\}$$

Tile index set, denoted by T^n, is the set of all tile indices whose tiles are non-empty

[2] All vectors in this paper are column vectors, although we use tuples to denote them in the text.

defined as follows:

$$T^n = \{\vec{t} \in Z^n | \exists \vec{i} \in I^n \ s.t.\ \vec{k} \cdot \vec{t} \leq \vec{i} \leq \vec{k} \cdot \vec{t} + \vec{k} - \vec{1}\}$$

or

$$T^n = \{\vec{t} \in Z^n | \exists \vec{i} \in Z^n \ s.t.\ \vec{k} \cdot \vec{t} \leq \vec{i} \leq \vec{k} \cdot \vec{t} + \vec{k} - \vec{1} \wedge C\vec{i} \leq \vec{c}\}$$

Notice that the above inequalities define a convex polyhedron in the $2n$-dimensional space spanned by $\vec{t}$ and $\vec{i}$. The tile index set T^n is an integer programming projection onto the n-dimensional subspace spanned by $\vec{t}$. Using the row-echelon algorithm described in [2], compilers can generate the loop bounds to scan T^n accurately. Let the loop bounds obtained to scan the tile index sets be $((TL_1, TU_1), \cdots, (TL_n, TU_n))$.

For example, the following DOACROSS loop nest

```
DO $(i_1, i_2)$ ((0, 7),(0,7))
    {
    $a(i_1, i_2) = 0.25 * ( a(i_1, i_2) + a(i_1 - 1, i_2) + a(i_1, i_2 - 1) + a(i_1 - 1, i_2 - 1) )$
    }
```

Suppose we use the size vector $\vec{k} = (2, 2) = \vec{2}$ to tile the index set of this loop nest. The tile with index $\vec{t}$ is

$$T_{\vec{t}} = \{\vec{i} \in Z^2 | \vec{0} \leq \vec{i} \leq \vec{7} \wedge \vec{2} \cdot \vec{t} \leq \vec{i} \leq \vec{2} \cdot \vec{t} + \vec{1}\}$$

The tile index set is

$$T^2 = \{\vec{t} \in Z^2 | \vec{0} \leq \vec{t} \leq \vec{3}\}$$

Loop tiling changes the execution order of the iterations to finish all iterations in a tile before executing the next one. The result of loop tile is the loop nest with $2n$ loops as follows:

```
$DO(t_1, \cdots, t_n)((TL_1, TU_1), \cdots, (TL_n, TU_n))$
    {
    $DO(i_1, \cdots, i_n)$ $((\max(L_1, t_1k_1), \min(U_1, (t_1 + 1)k_1 - 1)), \cdots,$
                        $(\max(L_n, t_nk_n), \min(U_n, (t_n + 1)k_n - 1)))$
        {
        $BODY$
        }
    }
```

The results of loop tiling the example in this section is

```
$DO(t_1, t_2)((0, 0), (3, 3))$
    {
    $DO(i_1, i_2)$ $((\max(0, 2t_1), \min(7, (2t_1 + 1)), (\max(0, 2t_2), \min(7, 2t_2 + 1)))$
        {
        $a(i_1, i_2) = 0.25 * ( a(i_1, i_2) + a(i_1 - 1, i_2) + a(i_1, i_2 - 1) + a(i_1 - 1, i_2 - 1) )$
        }
    }
```

Figure 2 illustrates the loop tiling of the example. The small circles represent the iterations of the loop nest. Each shaded square in the figure represents a tile. Arrows represents the data dependences between the loop iterations.

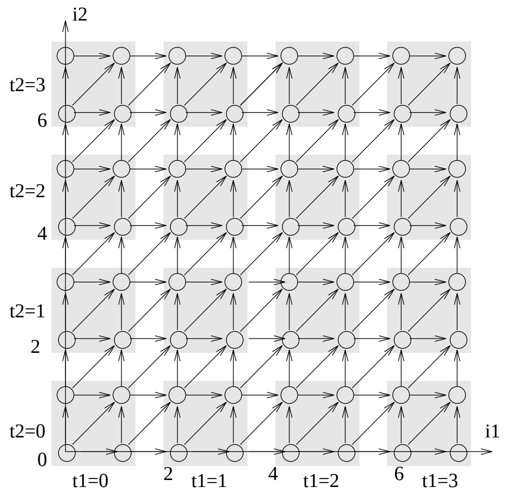


Figure 2. Tiled Loop Nest

Since the loop tiling changes the order of execution of the iterations, it is not always legal. However, if all the dependence distance vectors are non-negative, the rectangular tiling does not change the original execution order between any pair of iterations related by the dependences and, thus, is legal. If the distance vectors for the dependencies have negative elements, we can always skew the loop index set using unimodular transformation. An algorithm to calculate the unimodular matrix to skew the iteration space to makes the distances vectors of all dependencies non-negative can be found in [1]. In this paper, we assume that the distance vectors of all data dependencies are non-negative. That is, We have

$$\forall \vec{d} \in D \ : \ \vec{d} \geq \vec{0}$$

Each iteration $\vec{i}$ in the loop index set I^n belongs to only one tile in the tile index set T^n. For the rectangular tiling, the total function from I^n onto T^n can be expressed as follows:

$$\mathbf{t} : I^n \rightarrow T^n, \mathbf{t}(\vec{i}) = \vec{i} \ div \ \vec{k}$$

where opertator *div* applies to the corresponding elements of $\vec{i}$ and $\vec{k}$.

The purpose of loop tiling is to control the granularity of the parallel execution of the loop nest. The computation of the loop iterations of a tile is meant to be executed sequentially by a single processor. A processor issues send and receive commands for data communication only between the computations. The message size is determined by the tile size.

To execute the nested loop in parallel, we need to allocate and schedule the tiles to parallel processors. The basic ideas in the so-called chain-based scheduling are as follows:

1. To overlap the computation and data communication, each processor needs to be allocated sequences of tiles, called *chains*.

2. The chains of the tiles need to be allocated cyclically to parallel processors so that the full bandwidth of the multicomputer network can be utilized.

In the example of this section, we can allocate tiles with the same first tile index, *i.e.*,the tiles of tile indices $\vec{t} = (t_1, *)$, to the same processor. The chains are, then, allocated to processors cyclically as shown in the Figure 3. The thick arrows represent

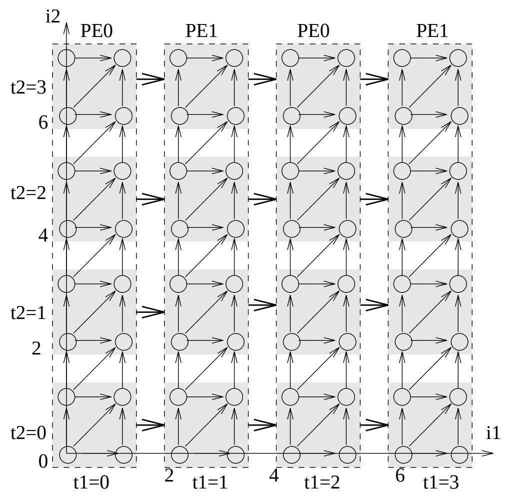


Figure 3. Chain-Based Scheduling of Nested Loop

the interprocessor communication due to the data dependences.

In general, parallel processors are assumed to be organized as an m-dimensional mesh $(1 \leq m < n)$. Formally, the *processor space* is

$$P^m = \{\vec{p} \in Z^m | \vec{0} \leq \vec{p} \leq \vec{P} - \vec{1}\}$$

where $\vec{P} = (P_1, \cdots, P_m)$ and $P_j > 1$ $(1 \leq j \leq m)$ is the size of the j-dimension. The total number of processors is $P_1 \cdots P_m$.

The tile allocation is a total function from tile index set T^n to processor space P^m defined as follows:

$$\mathbf{a} : T^n \to P^m, \mathbf{a}(\vec{t}) = \vec{t}|_{1,m} \bmod \vec{P}$$

where $\vec{t}|_{1,m}$ is the subvector $\vec{t}$ composed of the first m elements of $\vec{t}$ and operator mod applies to the corresponding elements of $\vec{t}|_{1,m}$ and $\vec{P}$. In this paper, we use $\vec{z}|_{p,q}$ $(1 \leq p < q \leq n)$ to denote the subvector $(z_p, \cdots, z_q)$ of vector $\vec{z} = (z_1, \cdots, z_n)$.

The tiles allocated to the same processor are executed in the lexicographic order of the tile index $\vec{t}$. The parallel code for each processor $\vec{p} \in P^m$ is shown in Figure 4. For the example of this section, we have one-dimensional processor array P^1 and the size of the processor array is $P_1 = 2$. The parallel code for processor $p \in \{0, 1\}$ (*i.e.*,PE_0 or PE_1) is as follows:

```
DO $t_1 = p, 3, 2$
    DO $t_2 = 0, 3$
        $\boxed{\text{Receive and Unpack Messages;}}$
        DO $i_1 = 2t_1, 2t_1 + 1$
        DO $i_2 = 2t_2, 2t_2 + 1$
            $a(i_1, i_2) = 0.25 * ( a(i_1, i_2) + a(i_1 - 1, i_2) + a(i_1, i_2 - 1) + a(i_1 - 1, i_2 - 1) )$
        ENDDO
        ENDDO
        $\boxed{\text{Pack and Send Messages;}}$
    ENDDO
ENDDO
```

The parallel code in Figure 4 also shows that whenever a processor finishes the computation of a tile, it sends the data generated by it to other processors. Similarly, each processor has to receive the data it needs before it starts the computation of a new tile. In the next two sections, we discuss how to generate and consume these messages for data communication.

3 Data Communication

There are two code sections left undefined in Figure 4: Pack and Send Messages, and Receive and Unpack Messages. Given a loop nest parallelized as shown in Figure 4, we want to increase the message size and reduce the number of messages as much as possible to amortize the large startup overhead of message passing. In this section, we answer the following questions:

- How many messages does a processor needs to send (receive) after (before) the computation of each tile?
- How to generate code for packing and unpacking each message?

```
DO $t_1 = TL_1 + (p_1 - TL_1 \bmod P_1) \bmod P_1, TU_1, P_1$
⋮
DO $t_m = TL_m + (p_m - TL_m \bmod P_m) \bmod P_m, TU_m, P_m$
    DO $t_{m+1} = TL_{m+1}, TU_{m+1}$
    ⋮
    DO $t_n = TL_n, TU_n$
        $\boxed{\text{Receive and Unpack Messages;}}$
        DO $i_1 = \max(L_1, t_1k_1), \min(U_1, t_1k_1 + k_1 - 1)$
        ⋮
        DO $i_n = \max(L_n, t_nk_n), \min(U_n, t_nk_n + k_n - 1)$
            ⋮
            ⋮
        ENDDO
        ⋮
        ENDDO
        $\boxed{\text{Pack and Send Messages;}}$
    ENDDO
    ⋮
    ENDDO
ENDDO
⋮
ENDDO
```

Figure 4. Chain-Based Code for Processor $\vec{p}$

It is well known that the output and anti data dependencies are caused by memory reuse of the data elements in programs. In distributed-memory multicomputers, the global arrays have to be implemented in local memories of parallel processors. If a data element is used in two different processors, it must have two copies in the two processors. In multicomputers, it makes no sense to enforce output and anti data dependences while physical memory locations of data elements on different processors have to be duplicated. In this paper, we assume that all the arrays and scalars in the loop nest are expanded [6] and every dependence in D is a flow dependences.

To simplify the presentation, we concentrate on the flow dependences raised from a single left-hand reference of an array, say array `A` and used $D_A \subseteq D$ to denote the set of dependences concerned. We use $\mathbf{f}$ to denote the subscript function of the left-hand side reference of D_A. For example, the elements of array `A` written by this left-hand side reference in tile $\vec{t}$ is $\{\mathbf{f}(\vec{i}) | \vec{i} \in T_{\vec{t}}\}$. Since all arrays are expanded, array `A` must have the same dimensionality as the index set and subscript function $\mathbf{f}$ is a one-to-one function.

In real programs, distance vectors are small non-negative integer vectors. When tiling a loop nest, the size vector, $\vec{k}$, is usually far larger than distance vectors. We assume that

$$\forall \vec{d} \in D_A \ : \ \vec{k} \geq \vec{d}$$

Subsequently, we can be assured that the data produced in a tile is only used by its neighboring tiles. Figure 5 shows the data communication for data dependence $\vec{d} = (2,1)$ between the tiles in a two-dimensional index set with tile size $\vec{k} = (4,4)$. Arrows in the figure show the flow data dependencies across tiles. The loop iterations in tile $\vec{t} = (0,0)$ producing the data to be used by its neighboring tiles $(1,0)$, $(0,1)$ and $(1,1)$, are marked with $W1 = W_{(0,0),(1,0),\vec{d}}$, $W2 = W_{(0,0),(0,1),\vec{d}}$, and $W3 = W_{(0,0),(1,1),\vec{d}}$, respectively. In general, a tile $\vec{t} \in T^n$ may produce data used by its $2^n - 1$ neighboring tiles $\vec{t} + \vec{b}$, where $\vec{b}$ is a vector in $B^n = B \times \cdots \times B$ with $B = \{0,1\}$.

In the following discussion, we first concentrate on the data communication between tiles for a single dependence $\vec{d} \in D_A$. Then we formulate the data sets for messages between processors. Finally, we show how to merge data sets of messages for multiple dependences in D_A.

We will focus on message packing and sending first. The data sets for message unpacking and receiving can be derived from those for message packing and sending.

Given a tile $\vec{t}$ in T^n and a distance vector $\vec{d}$ in D_A, we define the *remote write set* to the neighboring tile $\vec{t} + \vec{b}$ as follows:

Definition 2 (Remote Write Set) *The remote write set of tile $\vec{t}$ to the neighboring tile $\vec{t} + \vec{b}$ with respect to distance vector $\vec{d}$, denoted by $W_{\vec{t},\vec{b},\vec{d}}$, is the set of elements of array `A` written by tile $\vec{t}$ and read by tile $\vec{t} + \vec{b}$, i.e.*

$$W_{\vec{t},\vec{b},\vec{d}} = \{\mathbf{f}(\vec{i}) \mid \vec{i} \in T_{\vec{t}} \wedge \vec{i} + \vec{d} \in T_{\vec{t}+\vec{b}}\}$$

To enable compilers to generate code to pack and send messages for data communication between tiles, we need the inequality constraints for $W_{\vec{t},\vec{b},\vec{d}}$. From $\vec{i} \in T_{\vec{t}}$ and

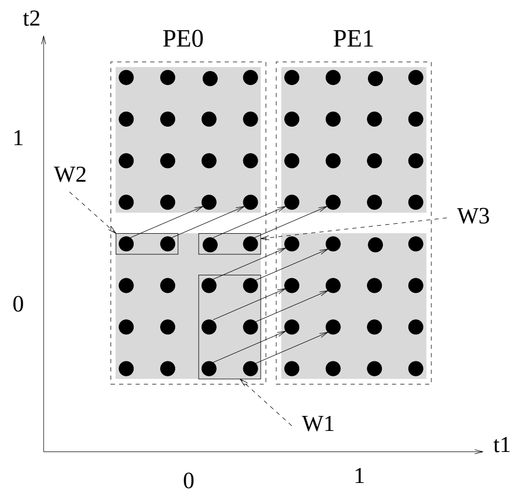


Figure 5. Remote Write Set for a dependence

$\vec{i}+\vec{d} \in T_{\vec{t}+\vec{b}}$, we have

$$\begin{aligned} \vec{0} &\leq \vec{i}-\vec{t}\cdot\vec{k} \leq \vec{k}-\vec{1} \\ \vec{0} &\leq (\vec{i}+\vec{d})-(\vec{t}+\vec{b})\cdot\vec{k} \leq \vec{k}-\vec{1} \end{aligned}$$

This leads to

$$\max(\vec{0},\vec{b}\cdot\vec{k}-\vec{d}) \leq \vec{i}-\vec{t}\cdot\vec{k} \leq \vec{k}-\vec{1}+\min(\vec{0},\vec{b}\cdot\vec{k}-\vec{d})$$

Recall that $\vec{b}$ is a vector in B^n with $B=\{0,1\}$ and $\vec{d}\leq\vec{k}$. Based on these facts, it is not difficult to show that the following is true:

$$\begin{aligned} \max(\vec{0},\vec{b}\cdot\vec{k}-\vec{d}) &= \vec{b}\cdot(\vec{k}-\vec{d}) \\ \min(\vec{0},\vec{b}\cdot\vec{k}-\vec{d}) &= (\vec{b}-\vec{1})\cdot\vec{d} \end{aligned}$$

The inequality constraints for $W_{\vec{t},\vec{b},\vec{d}}$ are summarized in the following lemma:

Lemma 1 *The inequality constraints for $W_{\vec{t},\vec{b},\vec{d}}$ are as follows:*

$$\begin{aligned} W_{\vec{t},\vec{b},\vec{d}} = \{\mathbf{f}(\vec{i}) \quad | \quad & C\vec{i}\leq\vec{c} \wedge C(\vec{i}+\vec{d})\leq\vec{c}\,\wedge \\ & \vec{b}\cdot(\vec{k}-\vec{d}) \leq \vec{i}-\vec{t}\cdot\vec{k} \leq \vec{k}-\vec{1}-(\vec{1}-\vec{b})\cdot\vec{d}\} \end{aligned}$$

In the example of Figure 5, we have $\vec{k} = (4,4)$ and $\vec{d} = (2,1)$. From Lemma 1, $M_{(0,0),(1,0),\vec{d}}$ is the set of data elements accessed by the iterations $\vec{i} \in I^n$ (*i.e.*,$\{\mathbf{f}(\vec{i})\}$) such that

$$\begin{pmatrix} 2 \\ 0 \end{pmatrix} \leq \vec{i} \leq \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

which is exactly the same as indicated in Figure 5.

The remote write sets described above provide a basic logical structure of potential data communication between tiles. Whether the interprocessor data communication for them is needed depends on whether the two neighboring tiles are allocated to different processors. If the tiles are allocated to the same processors, the data value can be passed by accessing the local memory of the processor. For example, if tiles are allocated to processors by columns as shown in Figure 5, the data communication represented by $W_{(0,0),(0,1),\vec{d}}$ does not need message passing. Moreover, the messages to the tiles that are allocated to the same processor can be merged. In the same example in Figure 5, the remote write sets $W1 = W_{(0,0),(1,0),\vec{d}}$ and $W3 = W_{(0,0),(1,1),\vec{d}}$ can be merged.

According to the tile allocation function $\mathbf{a}$ in the chain-based scheduling described in Section 2, tiles $\vec{t}$ and $\vec{t} + \vec{b}$ are allocated to the same processor if and only if $\vec{b}|_{1,m} = \vec{0}$[3]. That is,

$$\mathbf{a}(\vec{t}) = \mathbf{a}(\vec{t} + \vec{b}), \quad \textit{iff} \quad \vec{b}|_{1,m} = \vec{0} \tag{1}$$

Consider the remote write sets $W_{\vec{t},\vec{b}_1,\vec{d}}$ and $W_{\vec{t},\vec{b}_2,\vec{d}}$ of tile $\vec{t}$ to its two neighboring tiles $\vec{t} + \vec{b}_1$ and $\vec{t} + \vec{b}_2$. If $\vec{b}_1|_{1,m} = \vec{b}_2|_{1,m}$, the tiles $\vec{t} + \vec{b}_1$ and $\vec{t} + \vec{b}_2$ are allocated to the same processor:

$$\mathbf{a}(\vec{t} + \vec{b}_1) = \mathbf{a}(\vec{t} + \vec{b}_2)$$

The messages $W_{\vec{t},\vec{b}_1,\vec{d}}$ and $W_{\vec{t},\vec{b}_2,\vec{d}}$, therefore, can be merged.

Now we can define the messages for data communication between processors based on the above observation.

Definition 3 (Send Set) *Given a distance vector $\vec{d}$ and a vector $\vec{v} \in B^m$, the send set of tile $\vec{t}$ to its neighboring processor $\mathbf{a}(\vec{t} + \vec{v})$, denoted by $M^s_{\vec{t},\vec{v},\vec{d}}$, is the union of the remote write sets to the neighboring tiles allocated to the same processor:*

$$M^s_{\vec{t},\vec{v},\vec{d}} = \bigcup_{\vec{b}|_{1,m} = \vec{v}} W_{\vec{t},\vec{b},\vec{d}}$$

Note that vector $\vec{v}$ in the above definition is an m-vector from B^m where m is the number of dimensions of the processor array. When $\vec{v}$ is added to the n-vector $\vec{t}$, it is promoted to an n-vector by appending extra $n - m$ 0s to itself.

There are $2^m - 1$ send sets to the $2^m - 1$ neighboring processors for each tile. To obtain the inequality constraints for $M^s_{\vec{t},\vec{v},\vec{d}}$, consider the constraints in Lemma 1 on the

[3]In general, $\mathbf{a}(\vec{t}) = \mathbf{a}(\vec{t} + \vec{y})$ if and only if $\vec{y}|_{1,m} \bmod \vec{P} = \vec{0}$. Since we assumed $\vec{P} > \vec{1}$, we have $\vec{b}|_{1,m} \bmod \vec{P} = \vec{b}|_{1,m}$

j-th dimension with $m+1 \le j \le n$. If $b_j = 1$, we have

$$k_j - d_j \le i_j - t_j k_j \le k_j - 1$$

If $b_j = 0$, we have

$$0 \le i_j - t_j k_j \le k_j - d_j - 1$$

Merging these two cases gives the following inequality:

$$0 \le i_j - t_j k_j \le k_j - 1$$

The inequality constraints for $M^s_{\vec{t},\vec{v},\vec{d}}$, thus, can be summarized in the following theorem:

Theorem 1 *The inequality constraints for send set $M^s_{\vec{t},\vec{v},\vec{d}}$ is as follows:*

$$\begin{aligned} M^s_{\vec{t},\vec{v},\vec{d}} = \{\mathbf{f}(\vec{i}) \quad | \quad & C\vec{i} \le \vec{c} \wedge C(\vec{i}+\vec{d}) \le \vec{c} \wedge \\ & \vec{v}\cdot(\vec{k}|_{1,m} - \vec{d}|_{1,m}) \le \vec{i}|_{1,m} - \vec{t}|_{1,m}\cdot\vec{k}|_{1,m} \le \vec{k}|_{1,m} - \vec{1} - (\vec{1}-\vec{v})\cdot\vec{d}|_{1,m} \wedge \\ & \vec{0} \le \vec{i}|_{m+1,n} - \vec{t}|_{m+1,n}\cdot\vec{k}|_{m+1,n} \le \vec{k}|_{m+1,n} - \vec{1}\} \end{aligned}$$

We have described the data sets of communication for a single dependence in D_A. The data sets of multiple dependences in D_A should be further merged because all the data related to dependences in D_A are from the same left-hand reference. The merged message of tile $\vec{t}$ for all dependences in D_A to its neighboring processors $\mathbf{a}(\vec{t}+\vec{v})$, denoted by $M^s_{\vec{t},\vec{v}}$, can be defined as follows:

$$M^s_{\vec{t},\vec{v}} = \bigcup_{\vec{d}\in D_A} M^s_{\vec{t},\vec{v},\vec{d}}$$

Let $\vec{d}^{\max}$ and $\vec{d}^{\min}$ be the maximum and minimum distance vectors defined as follows:

$$\vec{d}^{\max} = \max_{\vec{d}\in D_A}(\vec{d})$$

$$\vec{d}^{\min} = \min_{\vec{d}\in D_A}(\vec{d})$$

We have

$$\vec{d}^{\min} \le \vec{d} \le \vec{d}^{\max}$$

for all $\vec{d}$ in D_A.

The inequality constraints for exact $M^s_{\vec{t},\vec{v}}$ can be complicated. Instead, we define a superset of it, denoted as $\widehat{M}^s_{\vec{t},\vec{v}}$, by taking the minimum of the lower bounds and the maximum of the upper bounds in each dimension as follows:

$$\widehat{M}^s_{\vec{t},\vec{v}} = \{\mathbf{f}(\vec{i}) \mid C\vec{i} \le \vec{c} \wedge C(\vec{i}+\vec{d}) \le \vec{c} \wedge$$

$$\vec{v} \cdot (\vec{k}|_{1,m} - \vec{d}^{\max}|_{1,m}) \leq \vec{i}|_{1,m} - \vec{t}|_{1,m} \cdot \vec{k}|_{1,m} \leq \vec{k}|_{1,m} - \vec{1} - (\vec{1} - \vec{v}) \cdot \vec{d}^{\min}|_{1,m} \wedge$$
$$\vec{0} \leq \vec{i}|_{m+1,n} - \vec{t}|_{m+1,n} \cdot \vec{k}|_{m+1,n} \leq \vec{k}|_{m+1,n} - \vec{1}\}$$

It is not difficult to prove the following theorem:

Theorem 2 $M^s_{\vec{t},\vec{v}}$ *is a subset of* $\widehat{M}^s_{\vec{t},\vec{v}}$, i.e.,$M^s_{\vec{t},\vec{v}} \subseteq \widehat{M}^s_{\vec{t},\vec{v}}$.

Compilers use $\widehat{M}^s_{\vec{t},\vec{v}}$ to generate code to pack and send messages to neighboring $2^m - 1$ processors.

The code generated for message packing and sending for all dependences in D_A are as follows:

```
begin
  for $\vec{v}$ in $B^m$ do
    pack a message according to the inequalities for $\widehat{M}^s_{\vec{t},\vec{v}}$;
    send the message to processor $\mathbf{a}(\vec{t} + \vec{v})$;
  endfor;
end
```

This code is to be put in the code section "Pack and Sending Messages" in Figure 4.

Since the processor executing $\vec{t} - \vec{v}$ sends message $\widehat{M}^s_{\vec{t}-\vec{v},\vec{v}}$ to its neighboring processor $\mathbf{a}(\vec{t})$, the later should receive the same data set and unpack it before the computation of $\vec{t}$. Given a tile $\vec{t}$, let the message to be received from the neighboring processor $\mathbf{a}(\vec{t} - \vec{v})$ ($\vec{v} \in B^m$) be denoted by $\widehat{M}^r_{\vec{t},\vec{v}}$. Obviously, we have

$$\widehat{M}^r_{\vec{t},\vec{v}} = \widehat{M}^s_{\vec{t}-\vec{v},\vec{v}}$$

By substituting $\vec{t} - \vec{v}$ for $\vec{t}$ in the inequality constraints for $\widehat{M}^s_{\vec{t},\vec{v}}$, we can have

$$\begin{aligned} \widehat{M}^r_{\vec{t},\vec{v}} &= \widehat{M}^s_{\vec{t}-\vec{v},\vec{v}} = \{\mathbf{f}(\vec{i}) \mid C\vec{i} \leq \vec{c} \wedge C(\vec{i} + \vec{d}) \leq \vec{c} \wedge \\ &\quad -\vec{v} \cdot \vec{d}^{\max}|_{1,m}) \leq \vec{i}|_{1,m} - \vec{t}|_{1,m} \cdot \vec{k}|_{1,m} \leq (\vec{1} - \vec{v}) \cdot (\vec{k}|_{1,m} - \vec{d}^{\min}|_{1,m}) - \vec{1} \wedge \\ &\quad \vec{0} \leq \vec{i}|_{m+1,n} - \vec{t}|_{m+1,n} \cdot \vec{k}|_{m+1,n} \leq \vec{k}|_{m+1,n} - \vec{1}\} \end{aligned}$$

The code to be put in the code section "Receive and Unpack Messages" in Figure 4 should be as follows:

```
begin
  for $\vec{v}$ in $B^m$ do
    receive a message from processor $\mathbf{a}(\vec{t} - \vec{v})$;
    unpack the message to the memory according to $\widehat{M}^r_{t,\vec{v}}$;
  endfor;
end
```

The messages passing described in this section is called *direct message passing*.

4 Indirect Message Passing

We have shown that for each dependence, a tile may have $2^m - 1$ messages to send to its neighboring processors, where m is the number of dimension of the processor array. We have also provided the inequality constraints for compiler to generate codes to pack and send these messages.

In this section, we present a scheme to further reduce the number of messages between processors. The scheme is called *indirect message passing* which reduces the number of messages from $2^m - 1$ to m.

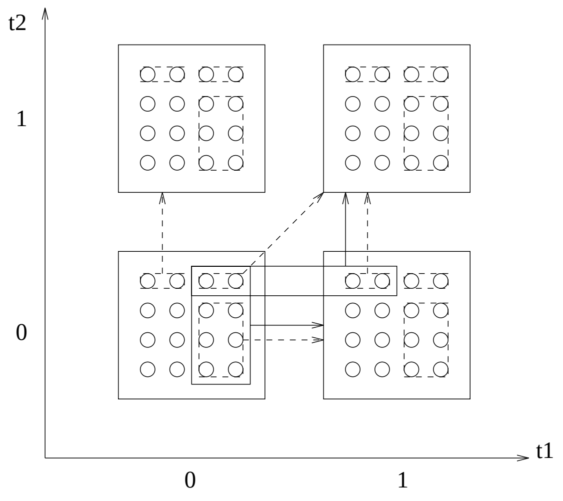


Figure 6. Indirect Message Passing

We use a simple example to illustrate the idea first. Suppose that we have a 3-dimensional loop index set ($n = 3$) and a 2-dimensional processor array ($m = 2$). Assume that the size of the loop index set is 8x8x8. The tile size vector is $\vec{k} = (4, 4, 4)$. Therefore, the size of the tile index set is 2x2x2. Assume that the size vector of the processor array is $\vec{P} = (2,2)$. According to the allocation function, each processor is allocated one chain. For instance, processor (0,0) is allocated chain (0,0) and chain (0,0) consists of tile (0,0,0) and tile (0,0,1). Figure 6 shows the top view of the 3-dimensional loop index set after tiling. The square boxes show the first tiles of the four chains. The circles in the square boxes represent the loop iterations. Assume that the dependence vector is $\vec{d} = (2,1,0)$. Therefore, tile (0,0,0) has three messages to send: $M^s_{(0,0,0),(1,0),\vec{d}}$ to processor (1,0), $M^s_{(0,0,0),(0,1),\vec{d}}$ to processor (0,1) and $M^s_{(0,0,0),(1,1),\vec{d}}$ to processor (1,1). All these messages are illustrated by dashed rectangular boxes and arrows in Figure 6. The idea is to merge $M^s_{(0,0,0),(1,1),\vec{d}}$ with $M^s_{(0,0,0),(1,0),\vec{d}}$ to be sent to processor (1,0) first. Then, $M^s_{(0,0,0),(1,1),\vec{d}}$ is merged with $M^s_{(1,0,0),(0,1),\vec{d}}$ and sent to processor (1,1) by processor (1,0). The direct passing of $M^s_{(0,0,0),(1,1),\vec{d}}$ from processor (0,0) to processor (1,1) is eliminated. Note that this does not cause any loss of parallelism because tile (1,1,0) will not start until it receives $M^s_{(1,0,0),(1,0),\vec{d}}$ from processor (1,0). The merged messages are illustrated by the solid rectangular boxes and arrows in Figure 6.

To simplify the presentation of indirect message passing in this section, we use the data sets to be received for each tile in direct message passing in the previous section. The *receive set* for a tile $\vec{t}$ from its neighboring processors $\mathbf{a}(\vec{t}-\vec{v})$ is defined as follows:

Definition 4 (Receive Set) *Given a distance vector $\vec{d} \in D_A$ and a vector $\vec{v} \in B^m$, the receive set of tile $\vec{t}$ from its neighboring processor $\mathbf{a}(\vec{t}-\vec{v})$, denoted by $M^r_{\vec{t},\vec{v},\vec{d}}$, is the send set by tile $\vec{t}-\vec{v}$ from processor $\mathbf{a}(\vec{t}-\vec{v})$ to processor $\mathbf{a}(\vec{t})$. That is,*

$$M^r_{\vec{t},\vec{v},\vec{d}} = M^s_{\vec{t}-\vec{v},\vec{v},\vec{d}}$$

Note that the m-vector $\vec{v}$ is promoted to an n-vector with $n-m$ 0s appended when it is subtracted from the n-vector $\vec{t}$ in the above notations.

By substituting $\vec{t}-\vec{v}$ for $\vec{t}$ in the inequality constraints in Theorem 1, we can have the inequality constraints for $M^r_{\vec{t},\vec{b},\vec{d}}$ summarized in the following theorem:

Theorem 3 *The inequality constraints for receive set $M^r_{\vec{t},\vec{v},\vec{d}}$ are as follows:*

$$\begin{aligned} M^r_{\vec{t},\vec{v},\vec{d}} = \{\mathbf{f}(\vec{i}) \quad | \quad & C\vec{i} \le \vec{c} \wedge C(\vec{i}-\vec{d}) \le \vec{c} \wedge \\ & -\vec{v}\cdot\vec{d}|_{1,m} \le \vec{i}|_{1,m} - \vec{t}|_{1,m}\cdot\vec{k}|_{1,m} \le (\vec{1}-\vec{v})\cdot(\vec{k}|_{1,m} - \vec{d}|_{1,m}) - \vec{1} \wedge \\ & -\vec{d}|_{m+1,n} \le \vec{i}|_{m+1,n} - \vec{t}|_{m+1,n}\cdot\vec{k}|_{m+1,n} \le \vec{k}|_{m+1,n} - \vec{d}|_{m+1,n} - \vec{1}\} \end{aligned}$$

4.1 Message Routing and Merging

Given a tile $\vec{t}$ and a dependence $\vec{d} \in D_A$, using the direct message passing described in Section 3 the processor executing $\vec{t}$ is supposed to receive $2^m - 1$ messages $M^r_{\vec{t},\vec{v},\vec{d}}$ from its neighboring processors $\mathbf{a}(\vec{t}-\vec{v})$. We distinguish those messages whose $\vec{v} \in B^m$ are elementary vectors and call them *elementary messages*:

Definition 5 (Elementary Message) *A message represented by a receive set, $M^r_{\vec{t},\vec{v},\vec{d}}$, is elementary if $\vec{v} = \vec{e}_j$ $(1 \le j \le m)$, where $\vec{e}_j$ is an elementary vector in B^m,* i.e.,$\vec{e}_1 = (1, 0, \cdots, 0)$, $\vec{e}_2 = (0, 1, \cdots, 0)$ *and* $\vec{e}_m = (0, 0, \cdots, 1)$.

The messages represented by $M^r_{\vec{t},\vec{v},\vec{d}}$ that are not elementary are called *forward messages*.

In the indirect message passing scheme, only elementary messages are directly passed between processors. The forward messages are merged with elementary messages and routed to the destination processor indirectly.

Consider a forward message $M^r_{\vec{t},\vec{v},\vec{d}}$ from tile $\vec{t} - \vec{v}$ at processor $\mathbf{a}(\vec{t} - \vec{v})$ to tile $\vec{t}$ at processor $\mathbf{a}(\vec{t})$. Assume that $\vec{v} = \vec{e}_{j_1} + \cdots + \vec{e}_{j_r}$ where $2 \le r \le m$ and $1 \le j_1 < \cdots < j_r \le m$. The message is to be forwarded to processor $\mathbf{a}(\vec{t} - \vec{v} + \vec{e}_{j_1})$ first, then to processor $\mathbf{a}(\vec{t} - \vec{v} + \vec{e}_{j_1} + \vec{e}_{j_2})$ and so on. After r steps, the message reaches the destination processor $\mathbf{a}(\vec{t} - \vec{v} + \vec{e}_{j_1} + \vec{e}_{j_2} + \cdots + \vec{e}_{j_r})$.

In each of these steps, the forward message $M^r_{\vec{t},\vec{v},\vec{d}}$ is merged with one of the elementary messages along the route. At step 1, it is merged with $M^r_{\vec{t}-\vec{v}+\vec{e}_{j_1},\vec{e}_{j_1},\vec{d}}$. At step 2, it is merged with $M^r_{\vec{t}-\vec{v}+\vec{e}_{j_1}+\vec{e}_{j_2},\vec{e}_{j_2},\vec{d}}$ and so on.

From the inequality constraints from Theorem 3, it is not difficult to see that for the forward message $M^r_{\vec{t},\vec{v},\vec{d}}$ with $\vec{v} = \vec{e}_{j_1} + \cdots + \vec{e}_{j_r}$ to be non-empty, elements $d_{j_1}, \cdots, d_{j_r}$ of $\vec{d}$ must be all greater than 0, *i.e.*,$d_{j_k} > 0$ $(k = 1, \cdots, r)$. For the same reason, all the elementary messages in the above algorithms, namely $M^r_{\vec{t}-\vec{v}+\vec{e}_{j_1},\vec{e}_{j_1},\vec{d}}$, $M^r_{\vec{t}-\vec{v}+\vec{e}_{j_1}+\vec{e}_{j_2},\vec{e}_{j_2},\vec{d}}$ and so on, are all non-empty. This means that forward messages are always merged with existing elementary messages and no extra messages are introduced.

4.2 Merged Receive and Send Sets

For a given tile $\vec{t}$ and a dependence vector $\vec{d}$, the elementary message along the q-th dimension merged with forward messages consists of three parts:

1. The q-th elementary message itself $M^r_{\vec{t},\vec{e}_q,\vec{d}}$ from processor $\mathbf{a}(\vec{t} - \vec{e}_q)$ to processor $\mathbf{a}(\vec{t})$.

2. The forward messages that reach the processor $\mathbf{a}(\vec{t})$ as their final destination. Let the forward message be $M^r_{\vec{t},\vec{v},\vec{d}}$. From the routing mechanism described above, we must have:

   - $v_q = 1$;
   - $\vec{v}|_{1,q-1} \ne \vec{0}$;
   - $\vec{v}|_{q+1,m} = \vec{0}$

3. The forward messages that pass processor $\mathbf{a}(\vec{t})$ as one of the intermediate processors. Let the forward message be from tile $\vec{t}_1$ at processor $\mathbf{a}(\vec{t}_1)$ to tile $\vec{t}_2 = \vec{t}_1 + \vec{v}$ at processor $\mathbf{a}(\vec{t}_2)$. According to the routing mechanism, we have $\vec{t}_1 + \vec{v}|_{1,q} = \vec{t}$ and $\vec{t} + \vec{v}|_{q+1,m} = \vec{t}_2$. Since $\mathbf{a}(\vec{t}) \ne \mathbf{a}(\vec{t} + \vec{v}|_{q+1,m})$ and hence, $\vec{t} \ne \vec{t}_2$, we have:

- $v_q = 1$;
- $\vec{v}|_{q+1,m} \neq \vec{0}$

The union of all the messages in the first and second parts can be expressed as

$$\bigcup_{v_q=1 \wedge \vec{v}|_{q+1,m}=\vec{0}} M^r_{\vec{t},\vec{v},\vec{d}} \tag{2}$$

The union of the messages of the third part can be expressed as

$$\bigcup_{v_q=1 \wedge \vec{v}|_{q+1,m}\neq\vec{0}} M^r_{\vec{t}+\vec{v}|_{q+1,m},\vec{v},\vec{d}} \tag{3}$$

Because the quantifying condition $\vec{v}|_{q+1,m} = \vec{0}$ in (2), the union of all the messages in parts 1 and 2 can be rewritten in a form similar to that of (3) as follows:

$$\bigcup_{v_q=1 \wedge \vec{v}|_{q+1,m}=\vec{0}} M^r_{\vec{t}+\vec{v}|_{q+1,m},\vec{v},\vec{d}} \tag{4}$$

Unifying expressions 3 and 4 gives rise to a single expression for the entire data set of the merged elementary message along the q-th dimension:

Theorem 4 (Merged Receive Message of $q-th$ Dimension) *Given a dependence $\vec{d}$, the merged receive set for tile $\vec{t}$ from the neighboring processor $\mathbf{a}(\vec{t}-\vec{e}_q)(1 \leq q \leq m)$, denoted as $IM^r_{\vec{t},q,\vec{d}}$, can be expressed as follows:*

$$IM^r_{\vec{t},q,\vec{d}} = \bigcup_{\vec{e}_q \leq \vec{v} \leq \vec{1}} M^r_{\vec{t}+\vec{v}|_{q+1,m},\vec{v},\vec{d}}$$

To obtain the inequality constraints for $IM^r_{\vec{t},q,\vec{d}}$, we use the inequality constraints for $M^r_{\vec{t},\vec{v},\vec{d}}$ in Theorem 3. We only need to consider the inequalities for the 1-st up to m-th dimensions. Consider the j-th dimension where $1 \leq j < q$ first. If $v_j = 1$, the constraint is

$$-d_j \leq i_j - t_j k_j \leq -1$$

If $v_j = 0$, we have

$$0 \leq i_j - t_j k_j \leq k_j - d_j - 1$$

Merging them gives rise to the following constraint:

$$-d_j \leq i_j - t_j k_j \leq k_j - d_j - 1$$

Now consider j-th dimension for $q \leq j \leq m$. If $v_j = 1$, we have

$$-d_j \leq i_j - (t_j + 1)k_j \leq -1$$

which can be transformed to

$$k_j - d_j \le i_j - t_j k_j \le k_j - 1$$

If $v_j = 0$, we have

$$0 \le i_j - (t_j + 0)k_j \le k_j - d_j - 1$$

Merging them gives rise to the following constraint:

$$0 \le i_j - t_j k_j \le k_j - 1$$

For the q-th dimension, we have

$$-d_q \le i_q - t_q k_q \le -1$$

Putting it all together, we have the following inequality constraints for the merged receive set:

Theorem 5 *The inequality constraints for the merged receive set $IM^r_{\vec{t},q,\vec{d}}$ is as follows:*

$$\begin{aligned} IM^r_{\vec{t},q,\vec{d}} = \{\mathbf{f}(\vec{i}) \quad | \quad & C\vec{i} \le \vec{c} \wedge C(\vec{i} + \vec{d}) \le \vec{c} \wedge \\ & -\vec{d}|_{1,q-1} \le \vec{i}|_{1,q-1} - \vec{t}|_{1,q-1} \cdot \vec{k}|_{1,q-1} \le \vec{k}|_{1,q-1} - \vec{d}|_{1,q-1} - \vec{1} \wedge \\ & -d_q \le i_q - t_q k_q \le -1 \wedge \\ & \vec{0} \le \vec{i}|_{q+1,n} - \vec{t}|_{q+1,n} \cdot \vec{k}|_{q+1,n} \le \vec{k}|_{q+1,n} - \vec{1} \wedge \\ & -\vec{d}|_{m+1,n} \le \vec{i}|_{m+1,n} - \vec{t}|_{m+1,n} \cdot \vec{k}|_{m+1,n} \le \vec{k}|_{m+1,n} - \vec{d}|_{m+1,n} - \vec{1}\} \end{aligned}$$

Using the same technique in Section 3 for merging $IM^r_{\vec{t},q,\vec{d}}$ for all $\vec{d}$ in D_A, we can have the merged receive set for D_A as follows:

$$\begin{aligned} \widehat{IM}^r_{\vec{t},q} = \{\mathbf{f}(\vec{i}) \quad | \quad & C\vec{i} \le \vec{c} \wedge C(\vec{i} + \vec{d}) \le \vec{c} \wedge \\ & -\vec{d}^{\max}|_{1,q-1} \le \vec{i}|_{1,q-1} - \vec{t}|_{1,q-1} \cdot \vec{k}|_{1,q-1} \le \vec{k}|_{1,q-1} - \vec{d}^{\min}|_{1,q-1} - \vec{1} \wedge \\ & -d_q^{\max} \le i_q - t_q k_q \le -1 \wedge \\ & \vec{0} \le \vec{i}|_{q+1,n} - \vec{t}|_{q+1,n} \cdot \vec{k}|_{q+1,n} \le \vec{k}|_{q+1,n} - \vec{1} \wedge \\ & -\vec{d}|_{m+1,n} \le \vec{i}|_{m+1,n} - \vec{t}|_{m+1,n} \cdot \vec{k}|_{m+1,n} \le \vec{k}|_{m+1,n} - \vec{d}|_{m+1,n} - \vec{1}\} \end{aligned}$$

We can also easily prove that

$$\bigcup_{\vec{d} \in D_A} IM^r_{\vec{t},q,\vec{d}} \subseteq \widehat{IM}^r_{\vec{t},q}$$

The inequalities for $IM^r_{\vec{t},q,\vec{d}}$ are used to receive and unpack the merged message for D_A from the neighboring processors $\mathbf{a}(\vec{t} - \vec{e}_q)$. The code to be generated for the section "Receive and Unpack Messages" in Figure 4 is

```
begin
   for $q = 1$ to $m$ do
      receive the merged message from processor $\mathbf{a}(\vec{t} - \vec{e}_q)$;
      unpack the message to the memory according to $\widehat{IM}^r_{\vec{t},q}$;
   endfor;
end
```

Similarly, we use $\widehat{IM}^s_{\vec{t},q}$ to denote the merged message for D_A to send to the neighboring processor $\mathbf{a}(\vec{t} + \vec{e}_q)$. Obviously, we have

$$\widehat{IM}^s_{\vec{t},q} = \widehat{IM}^r_{\vec{t}+\vec{e}_q,q}$$

The inequality constraints for $\widehat{IM}^s_{\vec{t},q}$ are

$$\begin{aligned}\widehat{IM}^s_{\vec{t},q} = \{\mathbf{f}(\vec{i}) \quad | \quad & C\vec{i} \le \vec{c} \wedge C(\vec{i} + \vec{d}) \le \vec{c} \wedge \\ & -\vec{d}^{\max}|_{1,q-1} \le \vec{i}|_{1,q-1} - \vec{t}|_{1,q-1} \cdot \vec{k}|_{1,q-1} \le \vec{k}|_{1,q-1} - \vec{d}^{\min}|_{1,q-1} - \vec{1} \wedge \\ & k_q - d^{\max}_q \le i_q - t_q k_q \le k_q - 1 \wedge \\ & \vec{0} \le \vec{i}|_{q+1,n} - \vec{t}|_{q+1,n} \cdot \vec{k}|_{q+1,n} \le \vec{k}|_{q+1,n} - \vec{1} \wedge \\ & -\vec{d}|_{m+1,n} \le \vec{i}|_{m+1,n} - \vec{t}|_{m+1,n} \cdot \vec{k}|_{m+1,n} \le \vec{k}|_{m+1,n} - \vec{d}|_{m+1,n} - \vec{1}\}\end{aligned}$$

The compiler-generated code to be put in the code section "Pack and Send Messages" in Figure 4 becomes

```
begin
   for $q = 1$ to $m$ do
      pack the merged message according to $\widehat{IM}^s_{\vec{t},q}$;
      send the message to processor $\mathbf{a}(\vec{t} + \vec{e}_q)$;
   endfor;
end
```

5 Experiment Results on AP1000

We have presented the mechanism to form large messages to amortize the startup overhead of message passing for nested DOACROSS loops executed on multicomputers. The basic techniques are loop tiling, chain-based scheduling (described in Sections 2 and 3) and indirect message passing (described in Section 4).

To investigate the effectiveness of these techniques, we first study the parallelism of chain-based DOACROSS loop nests on an ideal machine called *Parallel Random Access Machine* (PRAM). A PRAM is a parallel machine with an infinite shared random access memory. The data communication between processor is done through the shared memory. The time of memory access is assumed to be zero. In other words, the data communication in PRAM takes zero time and the speedup is restricted only by data dependences between tiles and the shape and size of the processor array. By comparing the speedup of DOACROSS on PRAM and real distributed-memory multicomputers, we can find the impact of data communication overhead and the effectiveness of loop tiling and indirect message passing.

Given a DOACROSS loop nest, a tile size vector $\vec{k}$ and a processor array P^m of PRAM as described in Section 2, we want to calculate the speedup of the parallel code in Figure 4 on the PRAM. To simplify the model, we assume that lower bounds of the n-dimensional DOACROSS loop nest are all zeros. The upper bound of the j-th loop is $N_j - 1$. According to the rectangular loop tiling (see Section 2), there are $S_1 \cdots S_n$ tiles where $S_j = N_j/k_j$ is the number of tiles along the j-th dimension of the tile index set T^n. It is assumed that N_j is an integer multiple of k_j for $1 \leq j \leq n$. We also assume that S_j is an integer multiple of P_j, the number processors along the j-th ($1 \leq j \leq m$) dimension of the processor array P^m.

It is obvious that there are $S_1 \cdots S_m$ chains. Each chain contains $S_{m+1} \cdots S_n$ tiles. We use $g = k_1 \cdots k_n$ to model the execution time of a tile. The execution time of each chain is

$$(S_{m+1} \cdots S_n)g$$

We define

$$L_{m+1} = S_{m+1} \cdots S_n$$

to be *the length of the chain*. The chains are allocated to parallel processors cyclically.

Suppose we have a 3-dimensional tile index set ($n = 3$) with ($S_1 = 4, S_2 = 9$ and $S_3 = 2$ to be allocated on a 2-dimensional ($m = 2$) processor array with $P_1 = 2$ and $P_2 = 3$. Figure 7, which is the top view of the 3-dimensional tile set, illustrates the processor allocation of the chains to the processor array. Each square in the figure represents the first tile of the chain and arrows represents the data dependences among them. The blank squares are the first tiles of the chains allocated to processor (0,0) and the darkest squares are the first tiles of the chains allocated to processor (2,3). The six processors will execute the chains in the box at the lower-left corner in Figure 7 first. Then the chains in the box on the right are executed and so on. In particular, processor (0,0) executes chain (0,0) first, and then chain (0, 3) and so on. After finishing chain (0,6), it comes back to execute chain (2,0) as illustrated in Figure 7. The starting time of the first tile of chain (0,0) is obviously 0. The start time of chain (0,3) by the same processor (0,0) is constrained by to the following to conditions:

1. Due to the data dependences along the second dimension of the chains, the starting time should be at least as large as $P_2 g = 3g$.

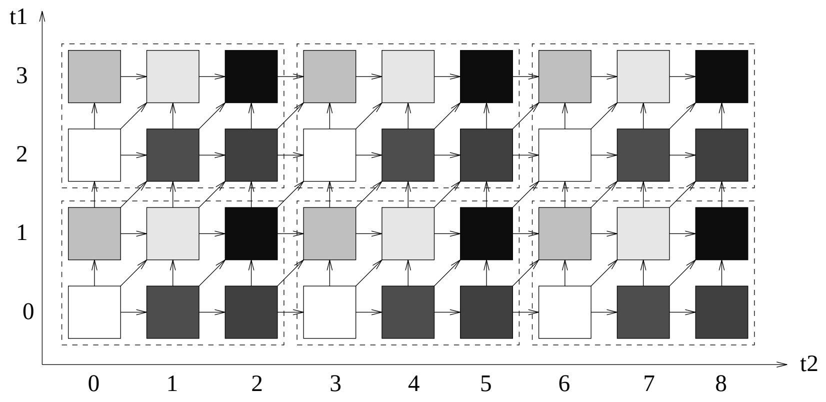


Figure 7. Chain-based Processor Allocation of the Example

2. Since processor (0,0) cannot start the first tile of chain (0,3) until it completes the entire chain (0,0), the start time should be at least as large as $L_3g = S_3g = 2g$

That is, the start time of chain (0,3) should be

$$\max(P_2, L_3)g$$

If P_2 was less than or equal to L_3, processor (0,0) can start chain (0,3) right after it finishes chain (0,0) and there is no time of busy-waiting between them. If $P_2 > L_3$, as the case in our example of Figure 7, processor (0,0) has to busy-wait for $(P_2 - L_3)g = g$ after each chain along the second dimension except the last one. Therefore, the total time for processor (0,0) to complete all the chains along the second dimension is

$$L_2g = (\frac{S_2}{P_2}L_3 + (\frac{S_2}{P_2} - 1)(P_2 - L_3)^+)g = 8g$$

where $(P_2 - L_3)^+$ is the notation for the *positive part* of the integer $P_2 - L_3$. The positive part of an integer x is defined by

$$x^+ = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Similarly, the start time of the first tile of chain (2,0) by processor (0,0) is constrained by: (1) the data dependences along the first dimension of the chains, and (2) the total time for processor (0,0) to complete the chains along the second dimension, L_2. It is

$$\max(P_1, L_2)g$$

For the same reason, the total time for processor (0,0) to complete its all chains along the first and second dimensions is

$$L_1 g = (\frac{S_1}{P_1} L_2 + (\frac{S_1}{P_1} - 1)(P_1 - L_2)^+)g = 16g$$

Consider the start time of the chains allocated to processor (2,3). Due to the data dependences of among the first tiles of the chains the start time of the first tile of a chain by processor (2,3) is always

$$(P_1 + P_2 - 2)g = 3g$$

later than that of the chain started by processor (0,0) in the same box in Figure 7. It is clear now that the total time to complete all chains of the loop nest by the six processors is

$$T_P = [L_1 + (P_1 + P_2 - 2)]g = 19g$$

The sequential execution time of the loop nest by one processor is

$$T_S = S_1 S_2 S_3 g = 72g$$

The speedup of the parallel execution is

$$S_P = T_S / T_P = 3.80$$

In general, the speedup of parallel execution over sequential execution can be calculated as follows:

1. Calculate

$$L_{m+1} = S_{m+1} \cdots S_n;$$

$$L_m = \frac{S_m}{P_m} L_{m+1} + (\frac{S_m}{P_m} - 1)(P_m - L_{m+1})^+;$$

$$\vdots$$

$$L_1 = \frac{S_1}{P_1} L_2 + (\frac{S_1}{P_1} - 1)(P_1 - L_2)^+$$

2. Compute speedup

$$S_P = \frac{S_1 \cdots S_n}{L_1 + (P_1 + \cdots P_m - m)}$$

If all the $(P_j - L_{j+1})^+ = 0$, $1 \leq j \leq m$, we can have the following closed formula for the speedup:

$$S_P = \frac{P_1 \cdots P_m}{1 + (P_1 + \cdots + P_m - m)(P_1 \cdots P_m)/(S_1 \cdots S_n)}$$

We parallelized the DOACROSS loop nest in Figure 1 by hand and run it on a Fujitsu AP1000 with different tile sizes and processor arrays. The AP1000 we use has 128 Sparc processors called *cells* connected by a two-dimensional (2D) mesh/torus network called

tile size	Fujitsu AP1000						PRAM
	direct			indirect			
	T_P^{ap}(sec)	$H(\%)$	S_P^{ap}	T_P^{ap}(sec)	$H(\%)$	S_P^{ap}	S_P^{pram}
4x8x2x2	67.424	27.85	92.2	61.312	20.66	101.4	127.8
4x8x4x4	54.362	10.16	114.4	54.036	9.62	115.1	127.3
4x8x8x8	52.910	6.22	117.5	52.853	6.12	117.6	125.3
4x8x16x16	55.122	4.34	112.8	55.197	4.81	112.6	117.9
4x8x32x32	67.599	3.50	92.0	67.676	3.60	91.9	95.3
8x16x2x2	54.435	10.28	114.2	53.650	8.97	115.9	127.3
8x16x4x4	51.329	3.34	121.1	51.002	2.71	121.9	125.3
8x16x8x8	53.477	1.40	116.3	53.309	1.08	116.6	117.9
8x16x16x16	65.440	0.31	95.0	65.030	-0.32	95.6	95.3
8x16x32x32	114.262	-0.95	54.4	113.873	-1.29	54.6	53.9

Table 1. Results on 2D (16x8) Processor Array

T net. The message passing primitives use T net for interprocessor data communication. The AP1000 Cell Operating Systems allows users to define a logical processor array of any number of dimensions, although the physical topology of T net is the 2D mesh/torus.

To get the sequential execution time of the program in Figure 1 on a single processor, we have to reduce the problem size from 128x128x128x128 to 32x32x32x32, because each sparc processor in the AP1000 has only 16Mbyte memory. The sequential execution time of this small size problem on a single processor is 24.285 seconds. We approximate the sequential execution time of the problem size 128x128x128x128 by multiplying this number with $128^4/32^4$, giving the approximate sequential execution time $T_S = 6216.96$ seconds.

We use two processor array configurations in our experiments: 16x8 and 8x4x4. Both contain 128 processors. For the 2D processor array (16x8), we fix the first and second components of the tile size vector $\vec{k}$ and change the third and fourth components from 2x2, 4x4, $\cdots$, up to 32x32. The fixed part of the size vector has two configurations: 4x8 and 8x16. We coded and run the parallel programs using both direct and indirect message passing. The results are shown in Table 1. T_P^{ap} in the table is the parallel execution time in seconds measured from the AP1000. S_P^{ap} is the speedup calculated by

$$S_P^{ap} = \frac{T_S}{T_P^{ap}}$$

The speedup on the PRAM, S_P^{pram}, calculated by the algorithm above is also listed in the table.

The communication overhead H listed in the table is calculated by

$$H = \frac{T_P^{ap} - T_P^{pram}}{T_P^{ap}}$$

where T_P^{pram} is the parallel execution time of the PRAM. Because the PRAM is an ideal parallel machine without communication overhead, the fraction defined above gives the percentage of the parallel execution time on the AP1000 caused by the data communication. The PRAM actually does not have parallel execution time, because it is not a real machine. For the purpose of comparison, it is calculated by

$$T_P^{pram} = \frac{T_S}{S_P^{pram}}$$

The speedup of the AP1000 with direct and indirect message passing as well as the speedup of the PRAM for 2D processor array are plotted in Figure 8. The communication overheads H are plotted in Figure 9.

For the 3D processor array (8x4x4), we fix the first three components of the tile size vector and change the fourth component from 2, 4, $\cdots$, up to 64. Table 2 lists the results for the two configurations of the first three components of the tile size vector: 4x8x8 and 8x16x16. The speedup and the communication overhead for the 3D processor array are

	Fujitsu AP1000						PRAM
	direct			indirect			
tile size	T_P^{ap}(sec)	H(%)	S_P^{ap}	T_P^{ap}(sec)	H(%)	S_P^{ap}	S_P^{pram}
4x8x8x2	66.235	26.44	93.9	62.818	22.44	99.0	127.6
4x8x8x4	60.738	19.53	102.4	59.093	17.29	105.2	127.2
4x8x8x8	58.637	16.12	106.0	57.605	14.62	107.9	126.4
4x8x8x16	57.543	13.43	108.0	56.992	12.59	109.1	124.8
4x8x8x32	58.236	12.35	106.8	57.898	11.84	107.4	121.8
4x8x8x64	98.924	9.05	62.8	99.030	9.15	62.8	69.1
8x16x16x2	56.477	11.80	110.1	55.871	10.84	111.3	124.8
8x16x16x4	55.688	8.34	111.6	55.314	7.72	112.4	121.8
8x16x16x8	57.572	7.07	108.0	57.312	6.65	108.5	116.2
8x16x16x16	62.298	6.21	99.8	62.074	5.87	100.2	106.4
8x16x16x32	72.305	5.51	86.0	76.542	10.74	81.2	91.0
8x16x16x64	117.604	4.58	52.9	117.396	4.41	53.0	55.4

Table 2. Results on 3D (8x4x4) Processor Array

plotted in Figure 10 and Figure 11, respectively.

From the performance data presented, we can have the following observations:

Speedup: From Figures 8 and 10, we first notice that the speedup on the ideal PRAM is affected by the tile size and shape. While S_P^{pram} approaches to 128 with small tile sizes, it can drop to as low as around 55 for the large tile sizes. In general, as the tile size increases, the difference between the speedups of the AP1000 and the PRAM decrease and the speedup of the AP1000 is restricted mainly by the limited parallelism available. As the tile size decreases, the data communication overhead increases and it pulls the speedup of the AP1000 down.

In each configuration, there is a tile size that gives the best speedup for the AP1000. The tile sizes that give the best speedup are marked by arrows in the plots. Interestingly, the both tile sizes for the best speedup on the 2D processor array are equal to 2^{11}. For the 3D processor array, the tile sizes for the best speedup are around 2^{12} to 2^{13}.

The highest speedup on the AP1000 observed is 121.9 with tile size 8x16x4x4 and indirect message passing on the 2D (16x8) processor array.

Communication Overhead: From Figures 9 and 11, we see that communication overhead decreases as the tile size increases. The communication overhead on 3D processor array is larger than that on 2D processor array. This is probably due to the two facts: (1) there are more messages in the 3D processor array than the 2D processor array and (2) the 3D processor array is actually simulated by the AP1000 Operating system on top of the 2D physical mesh/torus network. After all, the communication overhead is below 10% for the reasonably large tile sizes in most configurations.

Indirect Message Passing: When the tile size is small, the communication overhead of direct message passing is larger than that of indirect message passing. The gap between them increases as the tile size decreases. For some large tile sizes, the communication overhead of indirect message passing is larger than that of direct message passing (see tile size 4x8x16x16 on the 2D processor array and tile size 8x16x16x32 for the 3D processor array). This is because the large tile size causes more data to be forwarded in indirect message passing. After all, indirect message passing does help reduce the communication overhead in most configurations, but the reduction is limited.

There are a couple of anomalies in the speedups. We notice that, for tile sizes 8x16x32x32 and 8x16x16x16 (indirect message passing), the speedups of the AP1000 on the 2D processor array are larger than those of the PRAM by the small margin. That is why we observe the negative (small though) communication overhead in these configurations. This is probably due to the fact that we do not have the exact sequential execution of the program for the size 128x128x128x128.

We conclude this section with the following remarks:

- By combining and tuning loop tiling, chain-base scheduling and indirect message passing, compilers can generate efficient parallel codes on multicomputers with high speedup and low data communication overhead for DOACROSS loop nests.

- The parallelism of the chain-based parallel programs with tiling depends on the sizes and the shapes of the tile index set and the processor array. The algorithm to calculate the speedup of the PRAM can be used to search for the best configuration.

- The right tile size for the low communication overhead and high speedup depends on the machine used. On the AP1000, the right tile size for the DOACROSS loop nests like the one in Figure 1 is about 2^{11} to 2^{12}.

6 Conclusion

We have presented a compiler method that combines chain-based scheduling, loop tiling and indirect message passing to parallelize DOACROSS loop nests for distributed-memory multicomputers. The effectiveness of the method has been demonstrated by the experiment results on a 128-processor Fujitsu AP1000.

The experiment results show that the high speedup (about 120) can be obtained by tuning the tile size and processor array properly. By comparing the performances on the AP1000 and an ideal parallel machine without communication cost, we show that the data communication overhead in our method are very low (below 10%) with moderately large grain sizes.

We contribute the high speedup and the low communication overhead to the combined effect of chain-based scheduling, loop tiling and indirect message passing: chain-based scheduling makes it possible to overlap the data communication with the computation; loop tiling controls the grain size of the computation to amortize the message passing overhead and, finally, the indirect message passing further reduces the number of messages.

References

[1] P. Tang, "Chain-Based Scheduling: Part I – Loop Transformations and Code Generation," Department of Computer Science, The Australian National University, Technical Report TR-CS-92-09, June 1992.

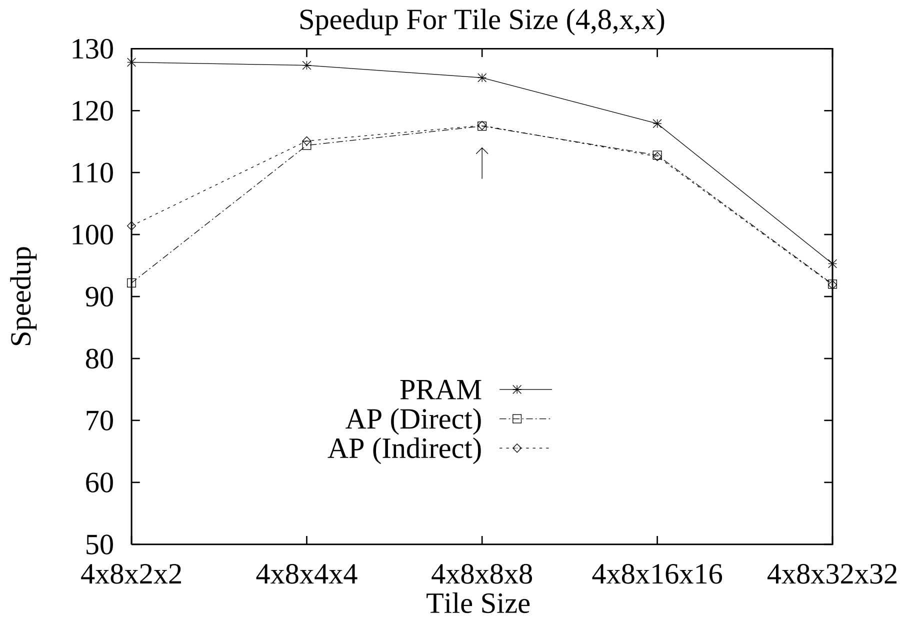
[2] C. Ancourt and F. Irigoin, "Scanning Polyhedra with DO Loops," in *Proceedings of the Third ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming*, Wiliamsburg, Virginia, April 21-24, 1991, pp. 39–50.

[3] M. E. Wolf and M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, October 1991.

[4] L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, February 1974.

[5] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimization for Supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, December 1986.

[6] P. Feautrier, "Array Expansion," in *Proceedings of the 1988 ACM International Conference on Supercomputing*, Malo France, July 1988, pp. 429–441.



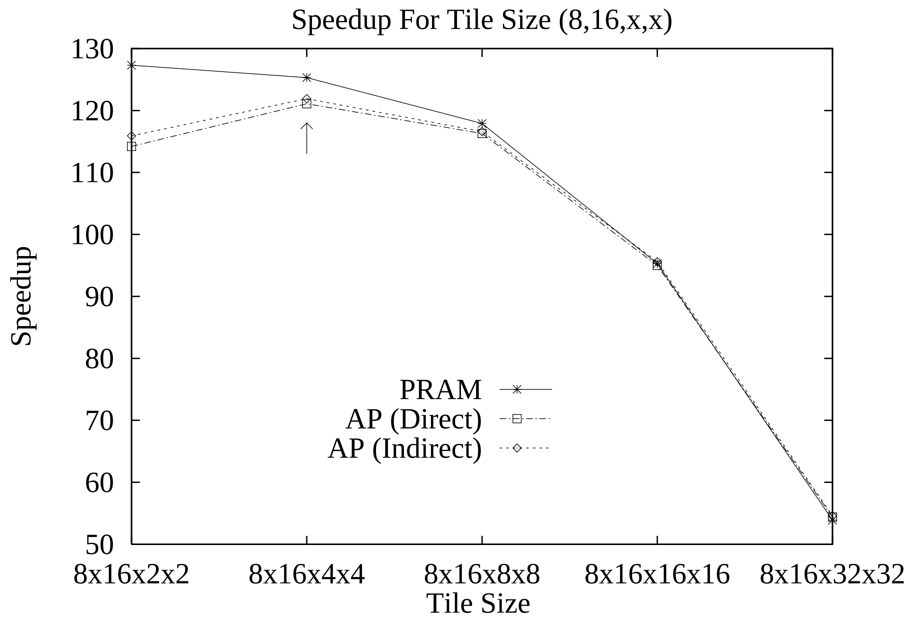
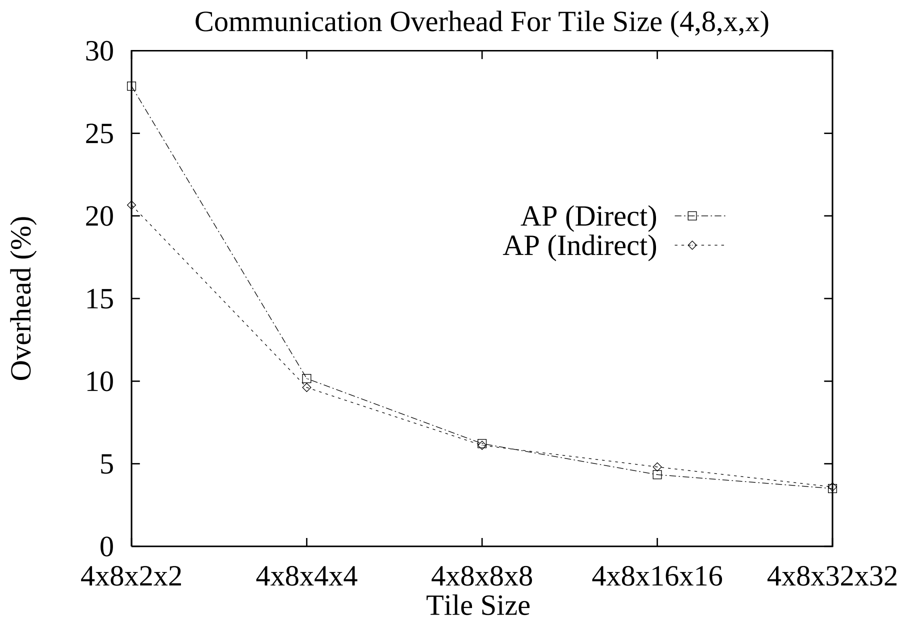


Figure 8. Speedup on 2D (16x8) Processor Array



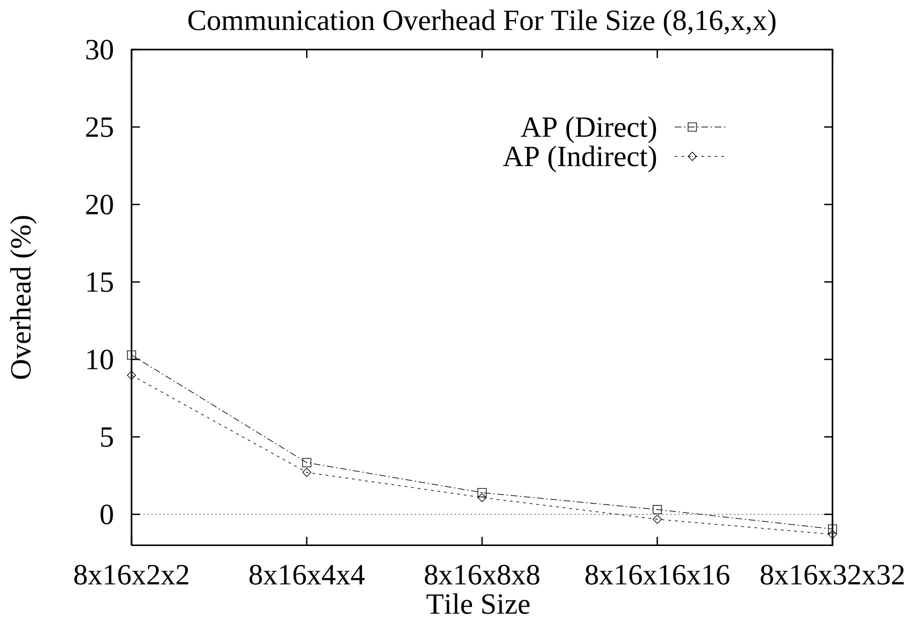
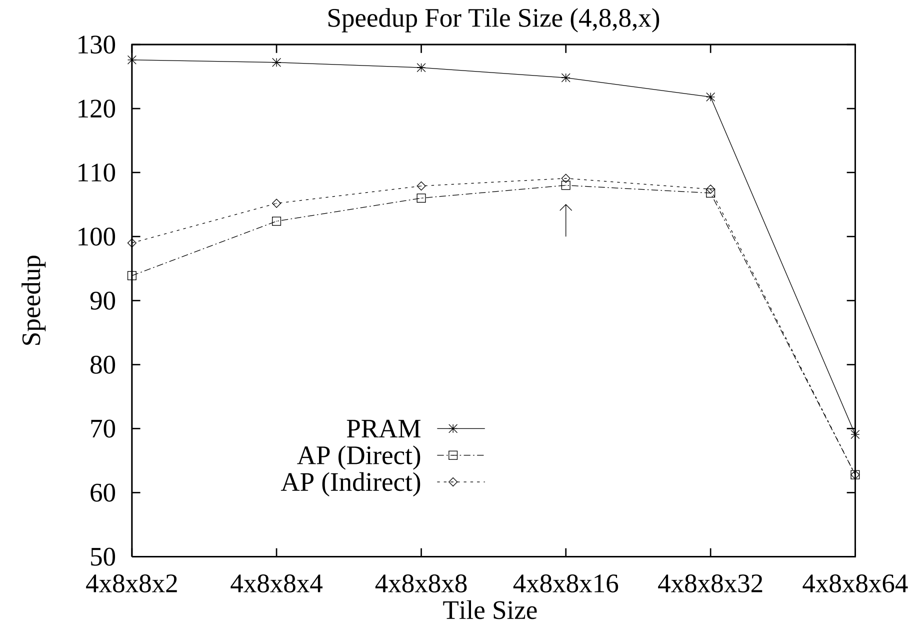


Figure 9. Communication Overhead on 2D (16x8) Processor Array



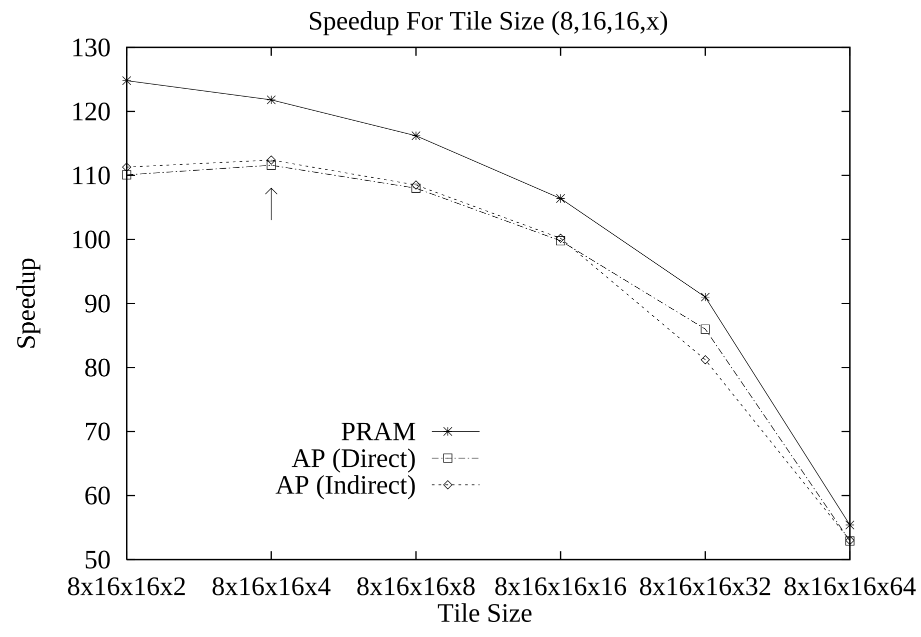
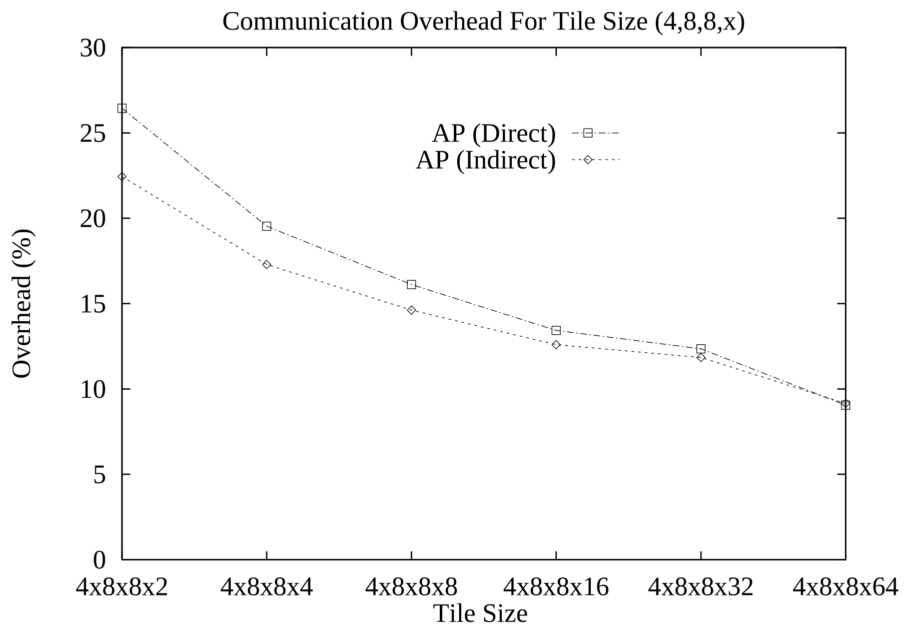


Figure 10. Speedup on 3D (8x4x4) Processor Array



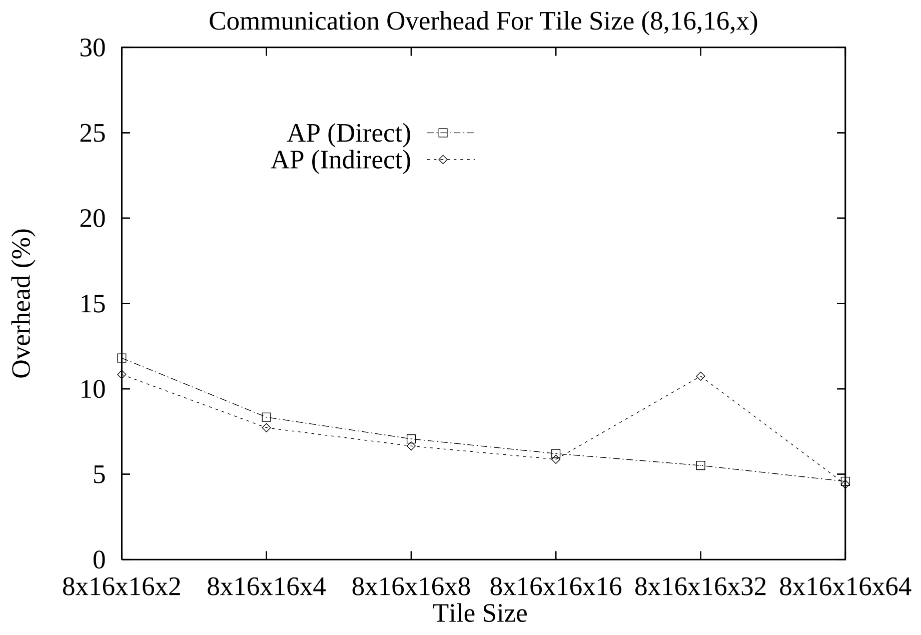


Figure 11. Communication Overhead on 3D (8x4x4) Processor Array