Iteration Space Tiling for Distributed Memory Machines

J. Ramanujam† and P. Sadayappan‡

†Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803-5901. Email: `jxr@max.ee.lsu.edu`

‡Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210-1277. Email: `saday@cis.ohio-state.edu`

Abstract

This paper addresses the problem of compiling nested loops for distributed memory machines. The relatively high communication start-up costs in these machines renders frequent communication very expensive. Motivated by this, we present a method for aggregating a number of loop iterations into **tiles** where the tiles execute atomically. Since synchronization is not allowed during the execution of a tile, partitioning the iteration space into tiles must not result in deadlock. We present an approach to determine the shape, size, allocation and scheduling of tiles for efficient parallel execution of 2-level tightly nested loops on distributed memory machines with given execution and communication costs.

1 Introduction

Distributed memory machines (multicomputers) offer high levels of flexibility, scalability and performance but programming these machines remains a difficult task. The message-passing paradigm employed in these machines makes program development significantly different from that for conventional shared memory machines. It requires that processors keep track of of data distribution and that they communicate with each other by explicitly moving data around in messages. As a result, parallelizing compilers for these machines have received great attention recently [2,5-7,12-15,21-24,27,29-32]. In addition, with current technology, the communication overhead is still at least an order of magnitude larger than the corresponding computation. The relatively high communication start-up costs in these machines renders frequent communication very expensive. This in turn calls for careful partitioning of the problem and efficient scheduling of the computations as well the communication.

Motivated by this concern, we present ways of partitioning nested loops and scheduling them on distributed memory machines with a view to **optimizing** the granularity of the resultant partitions. Nested loops are common in a large number of scientific codes and most of the execution time is spent in loops. In addition, they often lend themselves to simpler analyses. We present a method for aggregating a number of loop iterations into **tiles** where the tiles execute atomically – a processor executing the iterations belonging to a tile receives all the data it needs before executing any one of the iterations in the tile, executes all the iterations in the tile and then sends the data needed by other processors. Since synchronizations are not allowed during the execution of a tile, partitioning the iterations into tiles must not lead to deadlock. Given a perfectly nested loop to be executed on a multicomputer with given execution and communication costs, the tile shape and size have to be chosen so as to optimize the performance; in addition, the tiles must be assigned to processors to minimize communication costs and reduce processor idle times. We present an approach to determine the shape, size, allocation and scheduling of tiles for 2-level nested loops on distributed memory machines.

There have been tremendous advances in recent years in the area of intelligent routing mechanisms and efficient hardware support for inter-processor communication in message passing distributed memory machines [8]. These techniques reduce the communication overhead incurred by processor nodes that are incident on a path between a pair of communicating processors – this in turn greatly simplifies the task of mapping the partitions onto the processors. Therefore, we do not address the mapping problem here.

Section 2 discusses dependence analysis and other background material and presents a discussion of related work. In section 3 of this paper, we discuss the issues involved in tiling. Section 4 relates the concept of **extreme vectors** to deadlock-free tiling leading to section 5 where an algorithm to compute the extreme vectors in 2-dimensional iteration spaces is presented. Section 6 presents a method to choose deadlock-free tiles. Section 7 discusses **tile space graphs (TSGs)** and the scheduling and allocation of two dimensional tiles. In section 8, we discuss the determination of tile size so as to optimize the execution of nested loops on multicomputers. Section 9 summarizes the work and concludes with a discussion of further work.

2 Background

Good and thorough parallelization of a program critically depends on how precisely a compiler can discover the data dependence information. These dependences imply precedence constraints among computations which have to be satisfied for a correct execution. Many algorithms exhibit regular data dependences, *i.e.* certain dependence patterns occur repeatedly over the duration of the computation.

2.1 Data Dependence

Consider a loop structure of the form:

Example 1:
for $I_1 = l_1$ to u_1 do
  ...
    for $I_d = l_d$ to u_d do
      $S_x(\vec{I})$
      ...
      $S_y(\vec{I})$

---

where l_j and u_j are integer valued linear expressions involving $I_1, \ldots, I_{j-1}$ and $\vec{I} = (I_1, \ldots, I_d)$. $S_x, \ldots, S_y$ are assignment statements of the form $X_o = E(X_1, \ldots, X_k)$ where X_o is an output variable produced by expression E evaluated using some input variables $X_1, X_2, \ldots, X_k$.

The input variables of a statement are produced by other statements, and sometimes by the same statement at an earlier execution. Each computation is denoted by an index vector $\vec{I} = (I_1, \ldots, I_d)$.

Consider two statements S_x and S_y surrounded by perfectly nested loops. Note that S_x and S_y need not be two distinct statements. Data dependences are defined in [1, 3, 16, 17, 20, 28, 29]. A **flow dependence** exists from statement S_x to statement S_y if S_x computes and writes a value that can be subsequently (in sequential execution) read by S_y. Note that not all nest levels need to contribute to the dependence. A flow dependence implies that instances of S_x and S_y must execute as if some of the nest levels have to be executed sequentially. An **anti-dependence** exists between S_x and S_y if S_x reads a value that is subsequently written to by S_y. An **output dependence** exists between S_x and S_y if S_x writes a value which is subsequently written to by S_y. When dealing with loops (where the unit of scheduling is an iteration), dependence relations are represented as precedence constraints among the iterations of the loop.

2.2 Iteration Space Graph

Dependence relations are often represented in *Iteration Space Graphs* (ISG's); for an d-nested loop with index set $(I_1, I_2, \ldots, I_d)$, the nodes of the ISG are points on on a d-dimensional discrete Cartesian space and a directed edge exists between the iteration defined by $\vec{I_1}$ and the iteration defined by $\vec{I_2}$ whenever a dependence exists between statements in the loop constituting the iterations $\vec{I_1}$ and $\vec{I_2}$.

Many dependences that occur in practice have a constant distance in each dimension of the iteration space. In such cases, the vector $\vec{d} = \vec{I_2} - \vec{I_1}$ is called the **distance vector.** An algorithm has a number of such dependence vectors; the dependence vectors of the algorithm are written collectively as a **dependence matrix** $D = [\vec{d_1}, \vec{d_2}, \ldots, \vec{d_n}]$. In addition to the three types of dependence mentioned above, there is one more type of dependence known as **control dependence**. A control dependence exists between a statement with a conditional jump and another statement if the conditional jump statement controls the execution of the other statement. Of these data dependences, only flow dependence is inherent to the computation. In our analysis, we consider distributed memory machines where considering flow dependences alone is sufficient. Methods to calculate data dependence vectors can be found in [1, 3, 28, 29].

2.3 Related work in tiling and other memory optimizations

Chen et al. [6, 7] describes how Crystal, a functional language, addresses the issue of programmability and performance of parallel supercomputers. Gallivan et al. [9] discuss problems associated with automatically restructuring data so that it can be moved to and from local memories in the case of shared memory machines with complex memory hierarchies. They present a series of theorems that enable one to describe the structure of disjoint sub-lattices accessed by different processors, and how to use this information to make "correct" copies of data in local memories, and how to write the data back to the shared address space when the modifications are complete. Gannon et al. [10] discuss program transformations for effective complex-memory management for a CEDAR-like architecture with a three-level memory. In the context of hierarchical shared memory systems, Irigoin and Triolet [11] present a method, which divides the iteration space into clusters referred to as *supernodes,* with the goals of vector computation within a node, data re-use within a partition and parallelism between partitions. The procedure works from a new data dependence abstraction called dependence cones. King and Ni [12] discuss the grouping of iterations for execution on multicomputers; they present a number of conditions for valid tile formation in the case of two-dimensional iteration spaces. Mace [18] proves that the problem of finding optimal data storage patterns for parallel processing (the "shapes" problem) is NP-complete, even when limited to one- and two-dimensional arrays; in addition, efficient algorithms are derived for the shapes problem for programs modeled by a directed acyclic graph (DAG) that is derived by series-parallel combinations of tree-like subgraphs. Wolfe [27, 30] discusses a technique referred to as *iteration space tiling* which divides the iteration space of loop computations into *tiles* (or blocks) of some size and shape so that traversing the tiles results in covering the whole space. Optimal tiling for a memory hierarchy will find tiles such that all data for a given tile will fit into the highest level of memory hierarchy and exhibit high data reuse, reducing total memory traffic. In addition, in the context of loop unrolling, partitioning of iteration space graphs is discussed by Nicolau [19] and Callahan *et al.* [4].

2.4 Related work on compiling programs for multicomputers

Balasundaram and others [2] at CalTech are working on interactive parallelization tools for multicomputers that provide the user with feedback on the interplay between data decomposition and task partitioning on the performance of programs. Koelbel et al. [13, 14] address the problem of automatic process partitioning of programs written in a functional language called BLAZE given a user-specified data partition. A group led by Kennedy at Rice University [5] is studying similar techniques for compiling a version of FORTRAN 77 for the Intel iPSC/2 that includes annotations for specifying data decomposition; they show how some existing transformations could be used to improve performance. Rogers and Pingali [21] present a method which, given a sequential program and its data partition, performs task partitions to enhance locality of references. Zima et al. [15, 32] discuss SUPERB, an interactive system for semi-automatic transformation of FORTRAN 77 programs into parallel programs for the SUPRENUM machine, a loosely-coupled hierarchical multiprocessor.

3 Issues in tiling of iteration spaces

As mentioned earlier, the relatively high communication start-up costs in distributed memory machines renders frequent communication very expensive. For example, the message start-up time for Intel iPSC/2 in 250 μs for a packet where as the transmission cost per packet between neighboring nodes is about 10 μs; and access to local memory takes negligible time. Hence, we focus on collecting iterations together into *tiles* where each tile executes atomically with no intervening synchronization or communication; as a result, we are able to amortize the high message startup cost over larger messages at the expense of processor idle time.

Each tile defines an atomic unit of computation comprising of a number of iterations. All necessary data must be available before the beginning of execution of the tile and all data that are needed by some other processors for the execution of other tiles is available when the execution of the tile is complete – thus no synchronization or communication must be necessary during the execution of the tile. This imposes a constraint that the partitioning of the iteration space into tiles does not result in deadlock. This means that there are no cycles among the tiles. Thus the division into tiles must be compatible with the dependences on the nodes of the ISG. To keep code generation simple, it is also necessary that all tiles are identical in shape and size except near the boundaries of the iteration space.

Given the conditions on tiling, the problem is to choose a tile shape and size that minimizes execution time on a distributed memory machines with specified communication set-up and transfer costs and instruction execution rate. The approach involves:

1. determining valid partitioning of the iteration space (shape)
2. scheduling tiles
3. allocating tiles to processors to minimize communication and to reduce processor idle time
4. determining the granularity of tiles.

4 Extreme vectors and deadlock free tiling

We focus on tiles which are 2 dimensional subsets of 2-D iteration spaces. Tiles in 2-D spaces are defined by two families of parallel lines. Each such line is defined by a normal to it. Hence, tiles in 2-D iteration spaces are defined by a set of *two* vectors which are normals to the tile boundaries. Tiles defined in such a manner are parallelograms (except for those near the boundary of the iteration space). The shape of the tiles is defined by the families of lines and the size of the tiles is defined by the distance of separation between adjacent pairs of lines in each of the two families.

For tiles in 2-D iteration spaces to be legal, dependence vectors crossing a boundary between any pair of tiles must all cross from a given tile to the other *i.e.* the source of all the dependence vectors that cross the boundary must be in the same tile and their sink in the other. Any two vectors H_1 and H_2 define legal tiles if

$$H_i \cdot \vec{d_j} \geq 0, \qquad i = 1, 2 \tag{1}$$

for all d_j in the set of dependence vectors **or** if

$$H_i \cdot \vec{d_j} \leq 0, \qquad i = 1, 2. \tag{2}$$

The tile boundaries are aligned along the families of parallel lines in the 2-D case i.e. the tile boundaries are aligned along the normal to the vectors H_1 and H_2. The condition given here is from [11].

Based on results in linear and integer programming [25], the set of solutions to the system of linear inequalities in (1) is equivalent to the set of vectors that are expressible as a non-negative linear combination of two **extreme vectors** *i.e.,* the set of points that lie in the positive hull of two extreme vectors. In the case of 2-dimensional iteration spaces, the extreme vectors are a subset of the dependence vectors themselves. Thus, given a set of n distinct dependence vectors $(n \geq 2)$, $\vec{d_1}, \vec{d_2}, \ldots, \vec{d_n}$, one can always find two vectors say $\vec{d_1}$ and $\vec{d_2}$ (from among the $\vec{d_i}$) such that

$$c_i\vec{d_i} = a_i\vec{d_1} + b_i\vec{d_2} \tag{3}$$

where $i = 1, \ldots, n$, $a_i, b_i \geq 0$ and $c_i > 0$. The vectors $\vec{d_1}$ and $\vec{d_2}$ are the two **extreme vectors.** Without loss of generality, we assume $\vec{d_1}$ and $\vec{d_2}$ are linearly independent. If the dependence matrix has rank 2, *i.e.,* the iterations of the nested loop can not be executed in parallel without requiring proper synchronization (DOACROSS loop) [29], $\vec{d_1}$ and $\vec{d_2}$ will be linearly independent.

Condition (1) is only a sufficient condition for the following reason: it is always possible to choose tiles that are small enough - the extreme example being tiles consisting of exactly one iteration where there are dependences that have their source and sink in that iteration.

Figure 1: 2-dimensional iteration space and dependence vectors

5 Computing the extreme vectors

Here, we present an $O(n)$ algorithm for computing the extreme vectors, where n is the number of distance vectors and we assume that $n \geq 2$. It is based on the fact that in two dimensional iteration spaces, the first non-zero component in any distance vector must be non-negative and there are exactly two components to deal with. Therefore, distance vectors in 2-D iteration spaces are of one of the two following forms: $(0, \oplus)$ and $(+, *)$ where 0 stands for zero, $+$ represents positive integers, $*$ stands for all integers and $\oplus$ stands for non-negative integers. The distance vectors, therefore belong to the *first* and *fourth* quadrants (where the usual X-axis corresponds to the outer loop index variable i and the inner loop variable j is represented by the Y-axis). This is shown in Figure 1. The extreme vectors are the two vectors which have the maximum span from among all the dependence vectors *i.e.* every other dependence vector lies in the cone of the extreme vectors.

Let the components of each distance vector d_i be $(d_{i,1}, d_{i,2})$ where $d_{i,1}$ is the distance along the outer loop and $d_{i,2}$ is the distance along the inner loop. Find the ratio $r_i = \frac{d_{i,2}}{d_{i,1}}$ for all the dependence vectors (including the signs). The vectors with the highest and lowest values (one vector for each) are the extreme vectors. Note that if $d_{i,1} = 0$ for any d_i, then that vector is an extreme vector. If there is more than one vector with $d_{i,1} = 0$, we choose the vector with the smallest value of $d_{i,2}$ from among those. The largest and the smallest values among the r_i can be found in $O(n)$ time.

Consider the following loop:

Example 2:
for $i = 1$ to N
  for $j = 1$ to N
    $A[i,j] \leftarrow A[i,j-1] + A[i-1,j+2] + A[i-1,j] + A[i-1,j+1]$

The iteration space graph (ISG) is shown in Figure 2(a) and the distance vectors are shown in Figure 2(b). The loop has the following distance vectors $d_1 = (0,1)$, $d_2 = (1,-2)$, $d_3 = (1,0)$ and $d_4 = (1,-1)$; the ratios are $r_1 = \infty$, $r_2 = -2$, $r_3 = 0$ and $r_4 = -1$. r_1 has the highest ratio and and r_2 has the lowest ratio and therefore d_1 and d_2 are the extreme vectors. It is easy to verify that this is indeed the case.

However, in the case of higher dimensional iteration spaces, the extreme vectors need not be dependence vectors themselves [23, 24].

5.1 Transformational view of finding extreme vectors

The problem of finding extreme vectors in higher dimensions is discussed in [23] and in greater detail in the Ph.D. dissertation of the first author [24]. Here, we present the intuition behind the procedure in terms of 2-dimensional spaces. Intuitively, a transformation T is applied to the dependence matrix D such that every entry in the transformed dependence matrix is non-negative – in 2 dimensions, this is the minimum amount of "skewing" or "shearing" applied to the set of dependence vectors so that they "rotate" into the first quadrant. For the loop in example 2, the dependence matrix D is: (the columns are the dependence vectors):

$$D = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & -2 & -1 \end{pmatrix}$$

and transformation T is:

$$T = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$$

and the transformed dependence matrix is:

$$D^{+} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 2 & 0 & 1 \end{pmatrix}$$

6 Choosing deadlock-free tiles

The extreme vectors can always be used to form valid tiles. In addition, any non-negative linear combination of the extreme vectors can also be used as vectors (normals) defining the tile boundaries. Among the set of possible tilings, only the tiles defined by the extreme vectors lead to minimal communication. Choosing one of the tile boundaries aligned along an extremal vector results in communication only along the other tile boundary. Thus when we have n dependence vectors in two dimensional iteration spaces ($n \geq 2$), tile boundaries aligned along the extremal vectors $\vec{d_1}$ and $\vec{d_2}$ result in potential inter-processor communication across one of the tile boundaries alone for $\vec{d_1}$ and $\vec{d_2}$ and across both tile boundaries due to the rest of the dependence vectors (non-extremal vectors).

Figure 2(a): ISG of the loop in Example 2

Figure 2(b): Distance vectors of Example 2
Note: The extreme vectors are shown darker

Figure 2: Iteration Space Graph and Distance Vectors for Example 2

Figure 3: Tiles along $(0,1)$ and $(1,1)$ for example 3.
Note: Boundaries are shown in thicker lines. The slanted $(1,1)$ boundary leads to deadlock.

With a view to reducing communication, one might consider aligning the tile boundaries along some of the non-extreme dependence vectors. Consider the following loop:

Example 3:
for $i = 1$ to N
  for $j = 1$ to N
    $A[i,j] \leftarrow A[i,j-1] + A[i-1,j] + A[i-1,j-1] + A[i-2,j-2]$

In tiling the above loop, if tile boundaries were along the vectors $(1,1)$ and $(0,1)$, number of synchronization points (data communicated across the boundaries) is less than the case where the boundaries are aligned along $(0,1)$ and $(1,0)$ but the former defines illegal tiles. Figure 3 illustrates this. In fact, we show the following result:

Theorem 1 *Tiles of arbitrary size aligned along any of the dependence vectors other than the extreme vectors are* **illegal** *tiles.*

Proof: Given a set of n distinct dependence vectors, $\vec{d_1}, \vec{d_2}, \ldots, \vec{d_n}$, where $\vec{d_1}$ and $\vec{d_2}$ are linearly independent extreme vectors i.e., any dependence vector $\vec{d_i}$ can be written as:

$$c_i\vec{d_i} = a_i\vec{d_1} + b_i\vec{d_2}$$

where $i = 1, \ldots, n$, $a_i, b_i \geq 0$ and $c_i > 0$. If a tile boundary is oriented along a non-extreme $\vec{d_i}$, then the vector defining the family of parallel lines is orthogonal to $\vec{d_i}$ i.e.,

$$H_i \cdot \vec{d_i} = 0$$

Therefore,

$$H_i \cdot \left(a_i\vec{d_1} + b_i\vec{d_2}\right) = 0$$

Thus $H_i \cdot \vec{d_1}$ and $H_i \cdot \vec{d_2}$ must have opposite signs, which violates the condition for legality, *i.e.* either $H_i \cdot \vec{d_j} \geq 0$ or $H_i \cdot \vec{d_j} \leq 0$ for all d_j in the set of dependence vectors. Hence the result.

□

Note that if the extremal vectors are not linearly independent, then all the dependence vectors have the same unit vector along them – in which case tiling along that direction is always valid. The condition is a sufficient condition for generation of free size tiles (FST). Given the dependences, one can always choose tiles of a fixed size which violate the condition but are still valid.

7 Tile Space Graph (TSG)

Given the dependence vectors in the ISG and the tile boundaries, the precedence relationships (dependences) among the tiles is captured in the Tile Space Graph (TSG). We assume here that tile size is larger than the magnitude of any dependence vector *i.e* the tile size along each dimension is larger than the largest of corresponding components of the dependence vectors. This assumption leads to the following situation:

- The source and sink of any dependence vector that crosses a tile boundary are neighboring tiles.
- There are no two distance vectors d_1 and d_2 that cross the boundary between two tiles T_i and T_j such that the source of d_1 and the sink of d_2 lie in T_i while the sink of d_1 and the source of d_2 lie in T_j i.e. tiling does not result in deadlock.

If the components of the dependence vectors are large in magnitude, it is possible to choose small tiles, which are valid tiles but do not satisfy the condition that dependence vectors crossing a given tile boundary all do so in the same direction. For example, when the dependence vectors are $(0,1)$, $(1,0)$ and $(5,-1)$, tiles defined by the vectors $(0,1)$ and $(1,0)$ are valid as long as the length of the tile along the first dimension is less than or equal to 5, even though $(1,0)$ is not an extreme vector.

7.1 Scheduling of tiles

We are interested in choosing the size of the tile in order to optimize the execution time; hence we are interested in free size tiles (FSTs). In 2-D space with a regular ISG, any free size tile (FST) can only depend on at most 3 neighbors which satisfy exactly one of the following 3 conditions:

1. The tile depends on three neighbors exactly; in this case, two of those neighbors both depend on the third neighbor.
2. The tile depends on only on one neighbor.
3. All tiles are independent (in which case, the ISG defines a fully parallel loop)

Cases 2 and 3 correspond to the condition when the rank of the dependence matrix is not 2 for the case of 2-nested loops. Case 1 corresponds to the case when the dependence matrix has rank 2 and we will discuss this case. Figure 4 shows a sample TSG for 2-D iteration spaces with all the possible TSG vectors namely, $(0,1)$, $(1,0)$ and $(1,1)$. Since tiles are executed

Figure 4: Tile Space Graph for 2-nested loops.
Note: The tile space vectors $(0,1),(1,0)$ and $(1,1)$ are also shown.

atomically, the $(1,1)$ is automatically covered by the other two. Therefore, the tiles in the Tile Space Graph (TSG) have uniform dependence vectors $(0,1)$ and $(1,0)$. For scheduling of tiles, the scheduling hyperplane must satisfy the conditions of Lamport's result on wavefront execution [17, 26]. The scheduling hyperplane given by $(1,1)$ which is a line at angle of 45 degrees satisfies the conditions for a valid schedule. Figure 5 shows the wavefront schedule for 2-D tile spaces. If both tile space vectors are present, $(1,1)$ defines the optimal scheduling of tiles. The computations within a tile can be executed in lexicographic order. If each processor in the multicomputer have vector capability, then one can define wavefronts for the computations within each tile.

7.2 Allocation of tiles to processors

In general we may have many more tiles than the number of processors. The tiles must be allocated so that inter processor communication is minimized while the computation is load-balanced through time. Since the tile space has $(1,0)$ and $(0,1)$, choosing allocations of tiles along either direction "internalizes" all communication in that direction. Because of the presence of presence of both tile dependences $(0,1)$ and $(1,0)$, no more than one tile can be active along either a row or column. Thus, allocating the tiles to processors by row or column results in minimal processor idle time. Idle time is experienced only in the beginning and near the end of execution of the iterations. The interplay between load balance and communication depends on tile size and this is discussed in the next section.

Figure 5: Wavefront scheduling of tiles in 2-D iteration spaces.
Note: The wavefronts are shown using thicker lines.

8 Optimizing tile size

Let the computation be defined by an ISG of dimension $n_i \times n_j$ i.e., the outer loop (say indexed by i) runs from 1 to n_i and the inner loop (say indexed by j) runs from 1 to n_j. We make this assumption that n_j and n_i are comparable. Let the tiles be defined by parallelograms whose sides are l_i along the i axis and l_j along the j axis and let $l_i = l_j = l$ for simplicity of the expressions. This results in rhombic tiles. We will assume for now that the number of processors is unlimited and later show the result for a limited number of processors. Also, we assume for now that $n_j = n_i = n$. Finally, we assume that all tiles in a column of the Tile Space Graph are allocated to the same processor. Let the setup cost for a packet be s; let the cost of data transmission be w. See Figure 6 for description of various terms. (All these costs are normalized with respect to the cost of executing a single iteration). Thus the communication cost model is:

$$t_{comm} = s + w \times \text{length}$$

Let $c_i * l$ be the number of data items that may have to be sent across the vertical boundary (along the i axis in the TSG) and $c_j * l$ be the number of data items that may have to be sent across the horizontal boundary (along the j axis in the TSG). Since, we assume a column allocation of tiles, data must be communicated only across the vertical boundary (the boundary along the i axis.) The cost of sending $c_i * l_i$ values to another processor is

$$s + wc_il_i$$

Since columns of tiles are allocated to processors in a wrap-around fashion, if the network connecting the processors has Hamiltonian cycle, the above cost is accurate. Given that $l_i \times l_j$

Figure 6: Tile size determination in 2-D iteration spaces

nodes have to be executed in a tile, the cost of execution of the tile is:

$$\frac{n}{l_i}\left[l_i l_j + s + wc_i l_i\right] + \left(\frac{n}{l_j} - 2\right)\left[l_i l_j + s + wc_i l_i\right] + l_i l_j$$

The optimal tile size is one that minimizes this cost. In order to derive an expression for the optimal cost, (by solving for zeros of the derivative of the cost with respect to l) we assume that $l_i = l_j = l$ and that $\frac{n}{l} >> 1$. The time for execution (cost) is:

$$\text{Time,} \quad T = \frac{2n}{l}\left[l^2 + wcl + s\right]$$

Setting $dT/dl = 0$, we get

$$\begin{aligned} & \frac{dT}{dl} = 0 \\ \Longrightarrow \quad & \frac{2n}{l}\left[2l + wc\right] - \frac{2n}{l^2}\left[l^2 + wcl + s\right] = 0 \end{aligned}$$

assuming $s \neq 0$. This leads to

$$l^2 = s \quad \text{or} \quad l_{opt} = \sqrt{s}$$

If $s = 0$, then

$$\begin{aligned} & T = 2nl\left[l + wc\right] \\ \Longrightarrow \quad & l_{opt} = 1 \end{aligned}$$

Similar results can derived for an overlapped communication model.

Thus, the tile boundaries induce a Tile Space Graph for which we have identified that the wavefront schedule $(1, 1)$ is a valid scheduling hyperplane and in the case of dependence matrices whose rank is 2, this is also the optimal schedule when unlimited processors are available for loop execution.

9 Discussion

In this paper, we presented an approach to determine the shape and size of tiles for execution of nested loops on distributed memory machines; we presented sufficient conditions for legal tiles and a way to choose tile shapes for 2-D iteration spaces. In addition, we presented methods to allocate and schedule tiles on the processing nodes of a distributed memory machine and to determine the tile size that minimizes the execution time under our model. Given that $(1, 1)$ is a valid scheduling wavefront in the Tile Space Graph, the amount of parallelism (the number of tiles that can potentially be executed in parallel) in the execution of tiles increases and then decreases; thus it is conceivable that increasing the tile size along each dimension uniformly and then decreasing the same might lead to more efficient execution. Further research needs to be done in estimating the effect of varying the tile size on the performance of parallel execution of tiles.

References

[1] R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Trans. Programming Languages and Systems,* Vol. 9, No. 4, 1987, pp. 491-542.

[2] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer. An interactive environment for data partitioning and distribution. *Proc. 5th Distributed Memory Computing Conference (DMCC5),* Charleston, S. Carolina, April 1990.

[3] U. Banerjee, *Dependence Analysis for Supercomputing,* Kluwer Academic Publishers, Boston, MA, 1988.

[4] D. Callahan, J. Cocke and K. Kennedy, "Estimating Interlock and Improving Balance for Pipelined Architectures," *Journal of Parallel and Distributed Computing,* Vol. 5, No. 4, Aug. 1988, pp. 334-358.

[5] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing,* Vol. 2, Oct. 1988, pp. 151-169.

[6] M. Chen, "A Design Methodology for Synthesizing Parallel Algorithms and Architectures," *Journal of Parallel and Distributed Computing,* Vol. 3, Dec. 1986, pp. 461-491.

[7] M. Chen, Y. Choo and J. Li, "Compiling Parallel Programs by Optimizing Performance," *The Journal of Supercomputing,* 2, 1988, pp. 171-207.

[8] W. Dally and C. Seitz, "Deadlock-free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Computers,* Vol. C-36, No. 5, pages 547–553, May 1987.

[9] K. Gallivan, W. Jalby and D. Gannon, "On the Problem of Optimizing Data Transfers for Complex Memory Systems," *Proc. 1988 ACM International Conference on Supercomputing,* St. Malo, France, pp. 238-253.

[10] D. Gannon, W. Jalby and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations," *Journal of Parallel and Distributed Computing,* Vol. 5, No. 5, October 1988, pp. 587-616.

[11] F. Irigoin and R. Triolet, "Supernode Partitioning," in *Proc. 15th Annual ACM Symp. Principles of Programming Languages,* San Diego, CA, Jan. 1988, 319-329.

[12] C. King and L. Ni, "Grouping in Nested Loops for Parallel Execution on Multicomputers," *Proc. International Conf. Parallel Processing,* 1989, Vol. 2, pp. II-31 to II-38.

[13] C. Koelbel, P. Mehrotra and J. van Rosendale, "Semi-automatic Process Partitioning for Parallel Computation," *International Journal of Parallel Programming,* Vol. 16, No. 5, 1987, pp. 365-382.

[14] C. Koelbel and P. Mehrotra, "Semi-automatic Process Decomposition for Non-shared Memory Machines," in *Proc. 1989 International Conference on Supercomputing,* May 1989.

[15] U. Kremer, H. Bast, H. Gerndt and H. Zima, "Advanced Tools and Techniques for Automatic Parallelization," *Parallel Computing,* Vol. 7, 1988, pp. 387-393.

[16] D. Kuck, R. Kuhn, B. Leasure, D. Padua and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proc. ACM 8th Annual Symposium on Programming Languages,* Williamsburg, VA, Jan. 1981, pp. 207-218.

[17] L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM,* Vol. 17, No. 2, Feb. 1974, pp. 83-93.

[18] M. Mace, "Memory Storage Patterns in Parallel Processing," Kluwer Academic Publishers, Boston, MA, 1987.

[19] A. Nicolau, "Loop Quantization: A Generalized Loop Unwinding Technique," *Journal of Parallel and Distributed Computing,* Vol. 5, No. 5, Oct. 1988, pp. 568-586.

[20] D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM,* Vol. 29, No. 12, Dec. 1986, pp. 1184-1201.

[21] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference," *Proc. ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* Portland, OR, Jun. 1989, pp. 69-80.

[22] J. Ramanujam and P. Sadayappan, "A Methodology for Parallelizing Programs for Multicomputers and Complex Memory Multiprocessors," in *Proc. Supercomputing 89,* Reno NV, Nov. 13-17, 1989, pp. 637-646.

[23] J. Ramanujam and P. Sadayappan, "Tiling of Iteration Spaces for Multicomputers," *Proc. 1990 International Conference on Parallel Processing,* Vol 2, pages 179-186, August 1990.

[24] J. Ramanujam, "Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors," Ph.D. Thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, September 1990.

[25] A. Schrijver, *Theory of Linear and Integer Programming,* Wiley-Interscience series in Discrete Mathematics and Optimization, John Wiley and Sons, New York, 1986.

[26] M. Wolfe, "Loop Skewing: The Wavefront Method Revisited," *International Journal of Parallel Programming,* Vol. 15, No. 4, 1986, pp. 279-294.

[27] M. Wolfe, "Iteration Space Tiling for Memory Hierarchies," in *Parallel Processing for Scientific Computing,* G. Rodrigue (Ed.) SIAM, Philadelphia PA, 1987, pp. 357-361.

[28] M. Wolfe and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *International Journal of Parallel Programming,* Vol. 16, No. 2, 1987, pp. 137-178.

[29] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* Pitman Publishing, London and MIT Press, 1989.

[30] M. Wolfe, "More Iteration Space Tiling," in *Proc. Supercomputing 89,* Reno NV, Nov. 13-17, 1989, pp. 655-664.

[31] M. Wolfe, "Loop Rotation," Technical Report CS/E 89-004, Oregon Graduate Center, May 1989. Also to appear in a monograph to be published by Springer-Verlag.

[32] H. Zima, H. Bast and H. Gerndt, "SUPERB: A Tool for Semi-automatic MIMD/SIMD Parallelization," *Parallel Computing,* Vol. 6, 1988, pp. 1-18.